

- 大数据分析引擎Impala系统管理、性能优化与应用实践
- 怎么做大数据分析？大数据分析怎么应用在业务系统上？

开源大数据分析引擎 Impala实战

贾传青 著

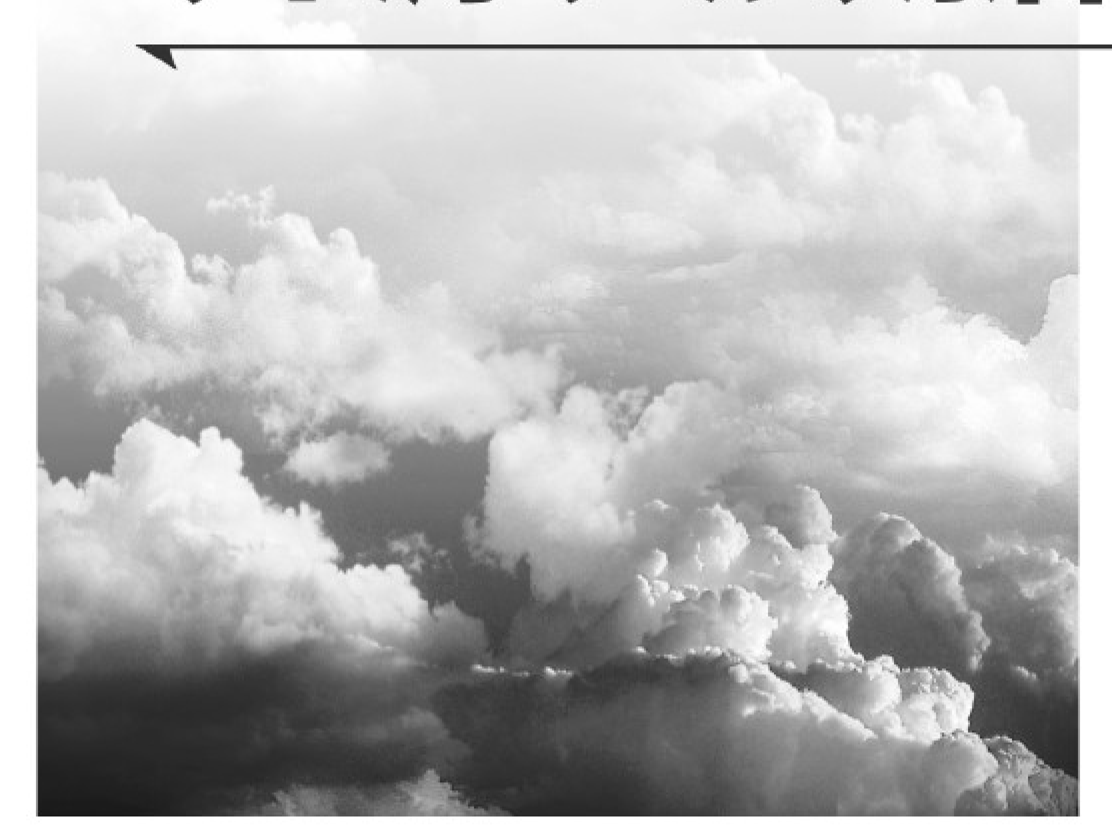
Big data

Cloud Computing

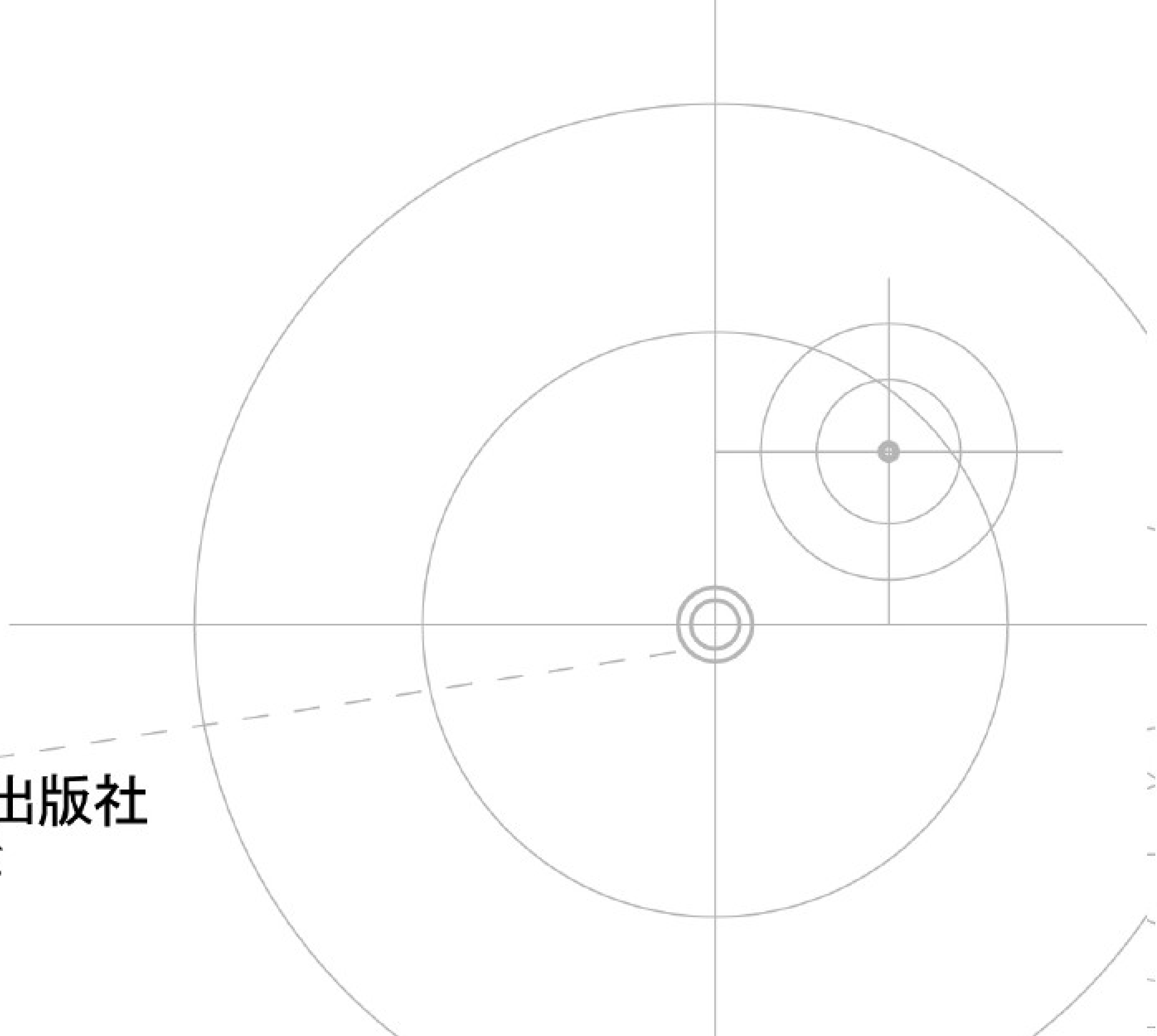
The internet of things

清华大学出版社

开源大数据分析引擎Impala实战



贾传青 著



清华大学出版社
北京

内 容 简 介

Impala 是 Cloudera 公司主导开发的新型查询系统，它提供 SQL 语义，能查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。Impala 1.0 版比原来基于 MapReduce 的 Hive SQL 查询速度提升 3～90 倍，因此，Impala 有可能完全取代 Hive。作者基于自己在本职工作中应用 Impala 的实践和心得编写了本书。

本书共分 10 章，全面介绍开源大数据分析引擎 Impala 的技术背景、安装与配置、架构、操作方法、性能优化，以及最富技术含量的应用设计原则和应用案例。

本书紧扣目前计算技术发展热点，适合所有大数据分析人员、大数据开发人员和大数据管理人员参考使用。

图书在版编目（CIP）数据

开源大数据分析引擎 Impala 实战 / 贾传青著.-北京 ：清华大学出版社，2015
ISBN 978-7-302-39002-2

I. ①开… II. ①贾… III. ①关系数据库系统 IV. ①TP311.138

中国版本图书馆 CIP 数据核字（2015）第 005181 号

责任编辑：夏非彼
封面设计：王　翔
责任校对：闫秀华
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张：**21.75　　**字　　数：**557 千字
版　　次：2015 年 3 月第 1 版　　**印　　次：**2015 年 3 月第 1 次印刷
印　　数：1~3000
定　　价：59.00 元

产品编号：057645-01

Cloudera 官方推荐序（中文）

大数据，作为目前工业界的主要技术趋势，定位于转化工业界的每一个细分市场，推动企业运用其数据开展业务的革命，并从根本上改变了支撑现代社会的 IT 基础架构。毫无疑问，大数据对中国意义重大，它给中国 IT 业的创新带来了巨大机会，没有其他任何一个国家比中国有更多的人口、更多的设备和更多的数据。

目前 Hadoop 是用于大数据的优选平台解决方案。作为 Hadoop 技术以及提供 Hadoop 解决方案的领导者，Cloudera 不仅提供经过了业界验证的 Hadoop 平台解决方案，也提供功能强大的工具帮助企业用户充分利用 Cloudera 企业版 Hadoop 解决他们的业务问题。Impala 就是 Cloudera 开发的众多强大工具之一。

Impala 是为了在 Hadoop 上实现低延迟的 SQL 查询而设计开发的，它原生地运行在 Hadoop/HBase 存储系统和元数据之上，因此它继承了 Hadoop 的灵活性、伸缩性和经济性，具有分布式本地化处理的特性以避免网络瓶颈，它与现有 Hadoop/CDH 的、基于工业标准的 SQL 接口兼容。它支持交互式 SQL，比最新版本的 Hive 快很多倍。由于 Impala 的这些优势，它受到了全球企业用户的热烈欢迎。

看到将为中国读者发布的这一本中文版 Impala 书籍，我非常欣喜，这无疑对中国用户更好地使用 Hadoop，解决他们的业务问题有很大帮助。因此，我要感谢所有为发布本书做出贡献的人们。最后，也要感谢广大读者对 Impala 的喜爱，以及你们在大数据——这一令人激动的 IT 发展方向上所做的贡献！

苗凯翔　博士

Cloudera 副总裁

Cloudera 官方推荐序（英文）

Big Data, as the next major trend in the industry today, is set to transform every market segment in the industry, revolutionize how enterprises will do their businesses using their data, and fundamentally shift the IT infrastructure underlying modern society. No doubt, Big Data is of great significance to China and it presents excellent opportunities to the IT industry in China for new innovations - no other country has more people, more devices, and more data than in China.

Hadoop is a platform solution of choice today for Big Data. As a leader of Hadoop and Hadoop-based solutions, Cloudera offers not only solid industry-proven Hadoop platform solutions, but also powerful tools to help enterprise users to fully leverage the Cloudera Enterprise Hadoop to solve their business problems. Impala is such a powerful tool, among many others created by Cloudera.

Impala is purpose-built for low-latency SQL queries on Hadoop, operating natively on Hadoop/HBase storage and metadata. As such it inherited the flexibility, scale, and cost advantages of Hadoop, and features distributed local processing to avoid network bottlenecks, and is compatible with SQL interface for existing Hadoop/CDH application based on industry standard SQL. It supports Interactive SQL which is typically many times faster than the latest Hive. Because of these advantages Impala has, it has become hugely popular among many enterprise users worldwide.

Now, I am very glad to see that a book focusing on Impala will be available in Chinese for the readers in China, which will definitely have a positive impact in helping users in China in better utilizing Hadoop and in solving the business problems they have. For this reason, I would like to thank all who have made contributions in making this book available in local language in China. Finally, I would like to thank the reader for your interest in Impala and for your contributions in this exciting direction of IT which we call Big Data!

Kai X. Miao（苗凯翔），Ph.D
Vice President, Cloudera Corporation

推荐序二

Impala是Hadoop生态圈中不可或缺的一个环节，它提供SQL语义，能够对HDFS和HBase中的PB级大数据进行交互式实时查询，从而弥补了Hive批处理的不足。本书是国内第一本Impala专业书籍，相信对您有益。

刘鹏

中国云计算专家咨询委员会副主任、秘书长

中国信息协会大数据分会副会长

推荐序三

Impala 起源于谷歌的 Dremel 大数据快速分析处理平台论文，由著名的大数据公司 Cloudera 开发并开源，在业内的知名度非常高，它立足于内存计算，从多迭代实时批量处理出发，兼顾数据仓库，是大数据系统领域的基于内存和 SQL 的快速处理分析计算平台。

近几年，大数据在如火如荼地发展着，随着谷歌开源了 Dremel 的论文，基于内存的计算分析技术逐渐走入大众的视野，谷歌为大家展示了一种超越 MapReduce 的数据分析之路，3 秒钟分析一个 PB 的数据已经不再遥不可及。但对于该论文的源代码实现寥寥可数，在这有限的开源代码中的翘楚者当属 Impala 和 Spark，但一直以来，无论是 Impala 还是 Spark，国内相关的文档和讨论都不甚全面。而本书的作者贾传青为大家带来了一本基于他多年的数据库和分布式工作的经验，可以说，这是 Impala 在目前国内最全面、最完整的技术讲解书籍。

随着内存分布式计算技术的发展，目前国内讨论该领域的社区也越来越多，但内存计算技术也存在几个问题，例如，Apache 的 Drill 项目迟迟没有开源。Spark 对于普通用户来说学习成本太高。Impala 则很好地设计了一个相对中庸的方案，在兼顾计算速度的同时，也照顾原有的 SQL 分析师和 Hive 的用户们。对于用户的快速上手开启大数据之旅十分方便快捷且简单实用。可以说，Impala 基于 HDFS、SQL 和 Hive，却又超越了 SQL、MapReduce 和 Hive。而它与经由传统数据库改造而来的分布式数据库如 Greenplum 等又有极大的差别。

这是国内第一本全面讲解 Impala 的书籍，既可以作为想快速搭建基于 Hadoop 的数据仓库的原数据库爱好者们的优秀参考书籍，又可以成为对 Spark 感兴趣的用户的架构理解入门书籍，因为这两种技术虽殊途但同源，Spark 虽效率更好，但架构更为复杂，开发难度也更高；而 Impala 恰恰为用户在复杂和简单之间兼顾了平衡。Impala 在国内的深入介绍和讨论的材料并不多，所以，作为第一本全面介绍 Impala 的书籍，其编写难度也是可想而知的。

非常荣幸能够为传青的这本书籍做序，同时也感谢他让我看到了 Impala 非常有特色的另一面。他是我认识的国内为数不多的研究 Impala 方面的专家，具有多年的数据库管理与开发经验，同时最近几年也在一直研究分布式和实时计算相关的最前沿技术，可以说，这本书既是他历史经验的总结，又是对他自己的一个新突破。

再次感谢传青邀请我为他的新书做序，在大数据的道路上，愿我们和读者们一起加油，向前。

向磊

EasyHadoop 与 phpHiveAdmin 作者

EasyHadoop 社区创始人，eXadoop 公司创始人

推荐序四

大数据应用的日渐增多为社会生活中的许多领域带来了积极变化。基于 Hadoop 的离线计算提供了强大的数据处理能力，但由于 Hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程，因此难以满足查询的交互性要求。能否有一项技术兼顾 DBMS/Hadoop 的混合优势呢？Cloudera 公司主导开发的 Impala 应运而生。有测试表明，Impala 的性能较 Hive 提高了 3~90 倍。

本书是国内第一本系统介绍 Impala 技术细节的书籍，对 Impala 的概念架构、SQL、资源管理、存储、分区进行了详细介绍。作者结合自身多年的 Oracle 和大数据研发经验，对 Impala 性能优化提出了自己的见解：通过数据对比可以看到良好的设计，以使计算性能有巨大提升。文章的最后，还根据设计原则给出了应用的具体案例。

作者贾传青执着于技术并乐于分享，他一直想写一本看着舒服的技术书籍。希望本书能够为有兴趣研究 Impala 的专业人员或学习者有所帮助。

郭刚

慧聪网 CTO

推荐序五

当下各类大数据技术的涌现已经成为了创新的源泉，让更多的想象成为可能。新一代开源大数据分析引擎 Impala 作为 Cloudera 在 Hadoop 领域的重要成员，其开源生态圈正在快速成长，技术应用前景广阔。贾先生与我在多个大数据技术领域有过深入交流，贾先生深厚的技术功底和严谨的钻研精神给我留下了深刻印象。非常高兴能看见贾先生的新著，这是我截至目前看到的第一本阐述 Impala 技术和应用最体系化的中文书籍。不要沉浸于讨论多大规模的数据才是“大数据”，本书将带领读者快速地掌握这个技术，打开大数据时代的窗户。大数据时代已经来临，先知先觉、先行先试者将先受益。

庄伟波

中信证券

推荐序六

鄙人从事电信行业数据库维护多年，和贾先生是多年的工作伙伴，很钦佩贾先生的才华，也很高兴他的新书能够出版。鄙人于 1999 年进入电信行业工作，从事信息化系统的维护工作，主管过本企业全省的核心营业、计费账务、经营分析等系统。拥有多年的 Oracle 数据库维护经验，是国内最早的 OCM 之一。

电信行业通过网格营销、市场监控、经营决策等提升用户体验，保有市场份额。而同时针对海量数据的分析，时效性又成为不容忽视的问题。天下武学，唯快不破，窃以为 IT 系统亦是如此。本书详细讲解了 Hadoop 生态系统中的实时分析引擎 Impala，相信能帮助每位读者快速掌握这一技术。

郭瑜敏
山西联通

推荐序七

我的博士研究方向是在可持续发展和网络组织背景下运用量化评价和数据分析方法建立区域经济发展模型。期间我沉淀了很多当今“大数据”分析过程中经常用到的数据分析和挖掘技能。近几年我负责了邮政业务量收数据仓库管理系统的建设工作。如何发挥数据仓库的存储和运算效率、持续提升技术投入成本的效益一直是我比较关注的问题。

Impala 作为基于 Hadoop 的实时计算技术可以直接通过 BI 产品进行展现，进行数据的查询和展示。在商业领域，如何发挥“大数据”的商业价值，帮助企业形成核心能力还没有形成一个成熟的框架模式。一些运用“大数据”技术的先行者们开展了积极的尝试，传青就是其中的一位专家。他的努力态度，所取得的成果和工作精神值得敬佩。

刁晓纯 博士
中国邮政
《实用数据分析》译者

作者序

写作背景

作为曾经的传统关系型数据库从业者，我们不仅需要了解数据库本身，还需要了解运行数据库的主机，存储数据库数据的仓库，读取数据库数据的中间件以及应用本身的特点。随着硬件的发展以及数据处理的细化，数据库技术从传统的基于磁盘的关系型数据库，向内存数据库、MPP 数据库不同的方向演进，数据库产品也从高大全向单一 RDBMS 吃遍天、短小精悍的方向发展。在架构时，我们必须根据应用的特点选择合适的数据库产品。

自 2009 年开始，笔者开始尝试使用基于 Hadoop 的技术来解决传统数据库无法线性扩展的问题。Hadoop 不能称之为“数据库”，也不能简单地称之为“应用”，而是介于数据库和应用之间的一种既能用于存储和处理数据，又能处理应用业务逻辑的一个混合体，我们通常称之为“数据平台”。Hadoop 虽在本质上解决了磁盘 IO 的扩展问题，但同时由于其基于磁盘（自 Hadoop 2.3 起支持缓存特性），因此对于某些实时性要求更高的任务无能为力，Impala 及其他的基于内存的运算技术应运而生。

Impala 的存储基于 HDFS，运算基于表的统计信息生成执行计划，具备资源管理功能，是最像传统数据库的大数据技术。笔者着手写作本书时 Impala 的最新版本为 1.3.1，而目前已演进至 2.1 版本，在 SQL 语法、安装、扩展性及性能方面进一步增强。

主要内容

工欲善其事，必先利其器，第 1 章手把手地为大家介绍如何离线搭建一个 Impala 环境。有了一个环境之后，我们可以暂时不考虑细节，先尝尝鲜使用一下它。第 2 章介绍如何在 Impala 上进行简单的数据加载、建表、查询等操作。作为 Impala 的管理者，仅仅能够简单使用它是远远不够的。第 3 章系统地介绍 Impala 的架构体系及各组件的作用。第 4 章是为 Impala 的使用者量身定做的，花费比较大的篇幅介绍了 Impala SQL、函数、UDF 等。任何一款数据库都会提供一个命令行工具，方便在没有图形界面的情况下，或者在 Shell 中进行调用，Impala 也不例外，第 5 章介绍 Impala 的命令行工具 Impala-shell。那如何有效地避免硬件资源的过载使用呢？当然是通过资源管理，第 6 章将详细介绍 Impala 的资源管理机制，另外也可以将 Impala 使用 YARN 来进行管理。第 7 章详细介绍了 Impala 底层支持的文件类型，基本囊括了 Hadoop 主流的文件类型。第 8 章介绍了 Impala 的分区机制。第 9 章介绍了 Impala 性能优化

的指导原则，以及优化过程中使用到的各项技术。第 10 章介绍了在企业应用中使用 Impala 时的设计原则及应用案例。

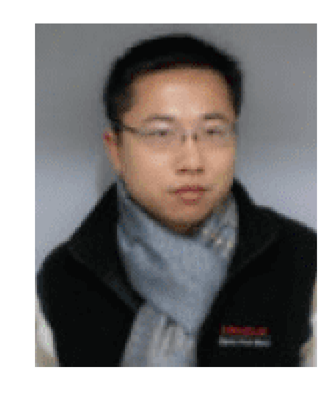
读者对象

- 内存计算技术初学者
- 数据库管理员及数据库开发人员
- Hadoop 及内存计算的运维工程师
- 开源软件爱好者
- 其他对大数据技术感兴趣的人员

致谢

在此感谢 Cloudera 的苗凯翔博士、Deborah Wiltshire、Yale Wang 对本书的认可。感谢我的好兄弟闫猛、付乐庆对我一直以来的鼓励。感谢我曾经服务过的客户们对我的信任。感谢我的家人和朋友们，你们是我不断努力的源动力。

作者简介



贾传青，数据架构师，Oracle OCM，DB2 迁移之星，TechTarget 特约作家，从数据库向大数据转型的先行者。曾服务于中国联通、中国电信、建设银行、PICC 等，目前供职于一家大数据解决方案提供商，致力于使用大数据技术解决传统数据库无法解决的问题。

作者

2015 年 1 月

目　录

第 1 章

Impala概述、安装与配置

Impala 是 Cloudera 公司发布的实时查询开源项目，从 1.0 版本开始，宣称比原来基于 MapReduce 的 Hive SQL 查询速度提升 3～90 倍，而且更加灵活易用。它提供 SQL 语义，能查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。已有的 Hive 系统虽然也提供了 SQL 语义，但由于 Hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程，难以满足查询的交互性。相比之下，Impala 的最大特点也是最大卖点就是它的快速。Impala 是高角羚的意思，这种羚羊主要分布在东非。

1.1 Impala 概述

Hadoop 的蓬勃发展得益于 Google 发表的关于 GFS 和 MapReduce 的论文。HDFS 和 MapReduce 都依据 Google 的论文为原型进行开发，HDFS 解决了分布式存储的问题，MapReduce 解决了分布式运算的问题。

使用者要很好地使用 Hadoop，需要能够熟练地使用 Java 或者其他语言编写 MapReduce 程序，这成为了传统数据库用户的瓶颈。为了解决这个问题，Hive 应运而生，通过 Hive 我们使用类似 SQL（HiveQL）的语句启动一个任务，由 Hive 将其翻译成一个 MapReduce 任务进行执行。这样大大降低了 Hadoop 的使用门槛。

MapReduce 非常适合用于批处理操作，对实时查询却无能为力。为了解决查询速度的问题，Cloudera 依据 Google 的 Dremel 为原型开发了跨时代的查询引擎：Impala。它抛弃了 MapReduce，使用更类似于传统的 MPP 数据库技术，大大提高了查询的速度。

Cloudera 公司网址如下，读者可以在这个网站上查阅到 Impala 应用的相关信息。

```
http://www.cloudera.com/
```

Impala 是一个开源的，能够进行快速查询的 Clouera 核心组件。本章我们将详细介绍 Impala 的安装过程。要安装 Impala 可以通过两个途径：

- 第一种方式使用 Cloudera Manager 进行安装。这是 Cloudera 推荐的安装方式。在 Cloudera Manager 4.8 或者之后的版本都可以对 Impala（1.2.1 或者更高的版本） 进行安装、配置、

管理、监控等操作。

- 第二种方式不使用 Cloudera Manager 手动安装。我们必须使用额外的检查步骤来确认 Impala 可以与其他 Hadoop 组件正确的交互，以及 Impala 集群本身的配置。

下面我们将使用第一种方式介绍 Impala 的安装，安装过程共分为 4 个部分，分别是 Cloudera Manager 安装准备，Cloudera Manager 及 CDH 安装，Hive 安装和 Impala 安装。

1.2 Cloudera Manager 安装准备

本书中演示示例使用的 Cloudera Manager（以下简称为 CM）的版本为 5.0.2，CDH 的版本也是 5.0.2，Impala 使用 CDH 版本对应的 1.3.1，如下图所示。

Cloudera Impala Version and Download Information

You can download the following releases of the Cloudera Impala product:

- **Impala 2.0.0** (latest 2.0.x, for CDH 4 or CDH 5)
- **Impala 1.4.0** (latest 1.4.x, for CDH 4 or CDH 5)
- **Impala 1.3.1** (latest 1.3.x, for CDH 4 or CDH 5)
- **Impala 1.2.4** (latest 1.2.x, for CDH 4)
- **Impala 1.2.3** (for CDH 4; also bundled with CDH 5 beta 2)
- **Impala 1.2.2**
- **Impala 1.2.1**
- **Impala 1.2.0 (Beta)** (for use with CDH 5 beta 1)
- **Impala 1.1.1** (latest 1.1.x)
- **Impala 1.0.1** (latest 1.0.x)

For installation instructions, see **Installing Impala (CDH 4 only)** .

介质下载地址：

```
CM5.0.2
http://archive-primary.cloudera.com/cm5/redhat/6/x86_64/cm/5.0.2/
CDH5.0.2
http://archive-primary.cloudera.com/cdh5/parcels/5.0.2/
```

需要将这两个 url 中的 mirrors 和 repodata 外的所有文件夹、子文件夹、文件全部下载到本地。介质需放在 HTTP 服务器 root 目录之下。

CM 5.0.2 安装下载的内容主要是 RPMS/x86_64 目录下的所有文件，如下图所示。

Index of /cm5/redhat/6/x86_64/cm/5.0.2/RPMS/x86_64

Name	Last modified	Size	Description
Parent Directory		-	
cloudera-manager-agent-5.0.2-1.cm502.p0.297.el6.x86_64.rpm	2014-06-11 18:09	3.7M	
cloudera-manager-daemons-5.0.2-1.cm502.p0.297.el6.x86_64.rpm	2014-06-11 18:09	315M	
cloudera-manager-server-5.0.2-1.cm502.p0.297.el6.x86_64.rpm	2014-06-11 18:09	8.0K	
cloudera-manager-server-db-2-5.0.2-1.cm502.p0.297.el6.x86_64.rpm	2014-06-11 18:09	9.6K	
enterprise-debuginfo-5.0.2-1.cm502.p0.297.el6.x86_64.rpm	2014-06-11 18:09	669K	
jdk-6u31-linux-amd64.rpm	2014-06-11 18:09	68M	
oracle-j2sdk1.7-1.7.0+update45-1.x86_64.rpm	2014-06-11 18:09	131M	

Apache/2.4.7 (Ubuntu) Server at archive-primary.cloudera.com Port 80

CDH5.0.2 安装下载的内容如下图所示。

Index of /cdh5/parcels/5.0.2

Name	Last modified	Size	Description
Parent Directory		-	
CDH-5.0.2-1.cdh5.0.2.p0.13-el5.parcel	2014-06-11 17:58	1.7G	
CDH-5.0.2-1.cdh5.0.2.p0.13-el5.parcel.sha1	2014-09-15 18:49	104	
CDH-5.0.2-1.cdh5.0.2.p0.13-el6.parcel	2014-06-11 17:56	1.7G	
CDH-5.0.2-1.cdh5.0.2.p0.13-el6.parcel.sha1	2014-09-15 18:48	104	
CDH-5.0.2-1.cdh5.0.2.p0.13-precise.parcel	2014-06-11 17:56	1.8G	
CDH-5.0.2-1.cdh5.0.2.p0.13-precise.parcel.sha1	2014-09-15 18:48	108	
CDH-5.0.2-1.cdh5.0.2.p0.13-sles11.parcel	2014-06-11 17:57	1.7G	
CDH-5.0.2-1.cdh5.0.2.p0.13-sles11.parcel.sha1	2014-09-15 18:49	107	
CDH-5.0.2-1.cdh5.0.2.p0.13-wheezy.parcel	2014-06-11 17:58	1.8G	
CDH-5.0.2-1.cdh5.0.2.p0.13-wheezy.parcel.sha1	2014-09-15 18:47	107	
manifest.json	2014-06-11 17:59	32K	

Apache/2.4.7 (Ubuntu) Server at archive-primary.cloudera.com Port 80

安装 CM 可以使用在线安装和离线安装的方式。在线安装方式也需要将安装包下载到本地再进行安装，所以对网络的带宽要求较高。而一般的企业的内外网是完全隔离的，所以本示例中我们将使用本地的 yum 源对 CM 进行安装，这种方式对于在企业中部署更为实用。

1. 安装环境（如下表所示）

主机名	IP	角色
hadoop-yum	192.168.0.10	yum 源服务器
hadoop-cm0	192.168.0.150	CM 管理服务器
hadoop-cm1	192.168.0.151	主节点
hadoop-cm2	192.168.0.152	备用主节点（本节点为主节点高可用提供，在实验中可以不安装）
hadoop-cs1	192.168.0.153	从节点
hadoop-cs2	192.168.0.154	从节点
hadoop-cs3	192.168.0.155	从节点

2. CM 支持的操作系统（如下表所示）

操作系统	版本	位数
兼容 Red Hat		
RHEL	5.7	64
	6.2	64
	6.4	64
CentOS	5.7	64
	6.2	64
	6.4	64
Oracle Unbreakable Linux	5.6	64
	6.4	64
SLEL		
SLES Linux Enterprise Server	11 SP1 或更高版本	64
Ubuntu/Debian		
Ubuntu	12.04	64
Debian	Wheezy(7.0,7.1)	64

在本示例的安装演示中使用了 CentOS 6.4 版本。

```
[root@hadoop-cm0 ~]# cat /etc/redhat-release
CentOS release 6.4
```

3. 支持的数据库

CM 的元数据需要存储在数据库中，CM 支持 MySQL、Postgresql、Oracle 等数据库，另外不同的组件对数据库的要求也不同，此处不再赘述。

本次安装使用自带的 Postgresql 数据库。

4. 支持的 JDK 版本

针对本次使用的 CDH 版本，JDK 的版本要求在 1.7.0_25 到 1.7.0_45 之间。

本次安装使用自带的 1.7.0_45。

```
-bash-4.1$ java -version
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
```

5. 支持的 IP 协议

CDH 只支持 IPv4 协议，不支持 IPv6。

```
[root@hadoop-cm0 ~]# lsmod |grep ipv6
```

执行上面指令，无输出为可行。

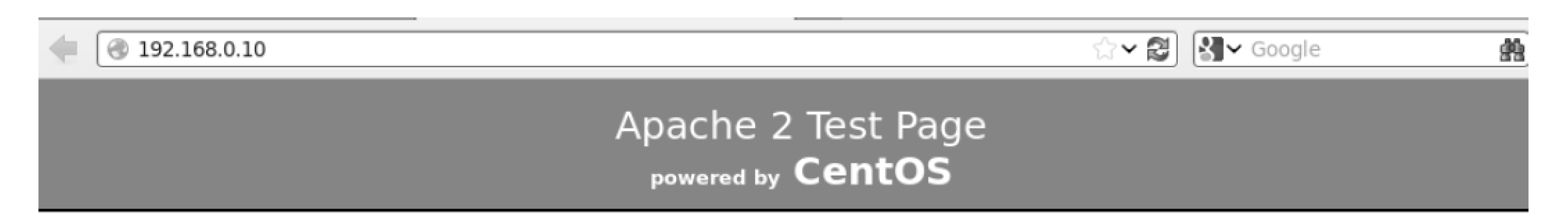
6. yum 源配置

在 hadoop-cm0 上启动 HTTP 服务，这里，CM 和 CDH 的介质需要放在 HTTP 服务器的 root 目录之下。

```
service httpd start
```

在浏览器地址栏中输入 http://192.168.0.10。

显示如下欢迎界面，如下图所示：



This page is used to test the proper operation of the Apache HTTP server after it has been installed. If you can read this page it means that the Apache HTTP server installed at this site is working properly.

If you are a member of the general public:

The fact that you are seeing this page indicates that the website you just visited is either experiencing problems or is undergoing routine maintenance.

If you would like to let the administrators of this website know that you've seen this page instead of the page you expected, you should send them e-mail. In general, mail sent to the name "webmaster" and directed to the website's domain should reach the appropriate person.

For example, if you experienced problems while visiting www.example.com, you should send e-mail to "webmaster@example.com".

If you are the website administrator:

You may now add content to the directory /var/www/html/. Note that until you do so, people visiting your website will see this page and not your content. To prevent this page from ever being used, follow the instructions in the file /etc/httpd/conf.d/welcome.conf.

You are free to use the images below on Apache and CentOS Linux powered HTTP servers. Thanks for using Apache and CentOS!



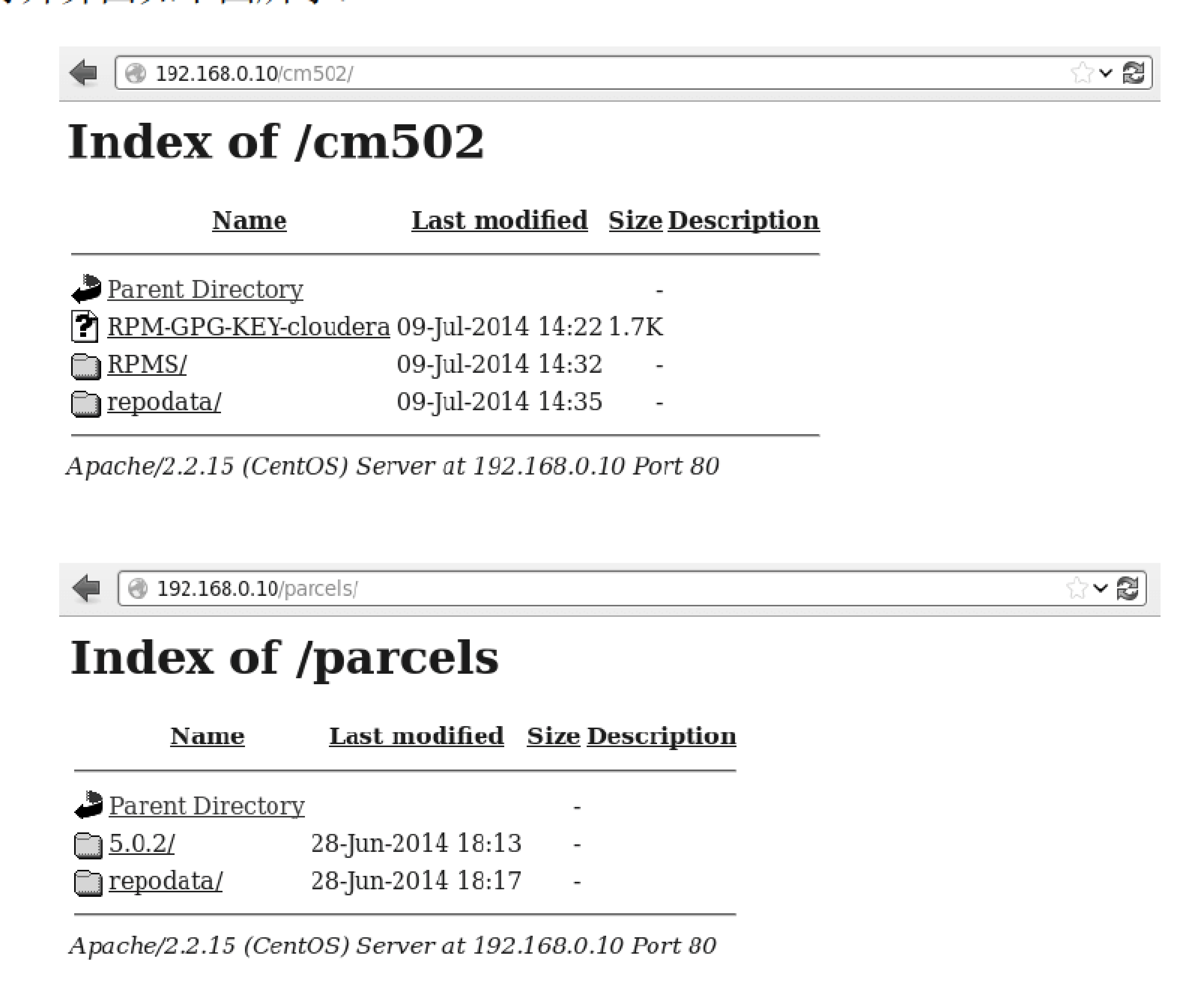


7. 对 RPM 文件建立索引

分别对 CM 和 CDH 的介质所在目录执行 createrepo。

```
[root@localhost cm502]# pwd
/soft/repo/cm502
[root@localhost cm502]# createrepo .
[root@localhost cdh502]# pwd
/soft/repo/parcels
[root@localhost cdh502]# createrepo .
```

创建完毕后会自动在相应的目录之下创建 repodata 的文件夹及相应的文件。通过浏览器可以访问介质，打开界面如下图所示：



创建 repo 文件并分发到各节点的/etc/yum.repo.d/下：

```
cloudera-cdh5.repo

[cloudera-cdh5]
name = Cloudera CDH, Version (Custom)
baseurl = http://192.168.0.10/parcels/
gpgkey = http://192.168.0.10/cm502/RPM-GPG-KEY-cloudera
gpgcheck = 1

cloudera-manager.repo

[cloudera-manager]
name = Cloudera Manager, Version 5.0.1
baseurl = http://192.168.0.10/cm502/
gpgkey = http://192.168.0.10/cm502/RPM-GPG-KEY-cloudera
gpgcheck = 1
```

另外，建议也为操作系统的安装介质建立一个本地 yum 源的 repo 文件，因为 CM 在安装过程中可能需要某些操作系统默认安装没有带的包。

```
base-centos6.repo

[base-centos6]
# Packages for Cloudera's Distribution for Hadoop, Version 5, on RedHat or CentOS
6 x86_64
name=CentOS6.5
baseurl=http://192.168.0.10/centos6
```

8. 关闭 SELinux 安全选项

修改如下配置文件：

```
[root@hadoop-cm0 selinux]# cat /etc/selinux/config
SELINUX=disabled
```

值得一提的是如果不修改配置文件只使用 setenforce 命令即使修改正确了，CM 仍然认为没有正确配置该安全选项。最好的办法就是修改配置文件并重启主机。

9. 关闭防火墙

执行如下命令：

```
[root@hadoop-cm0 selinux]#service iptables stop
```

关闭所有节点防火墙。

10. 对等性配置

本步骤配置 ssh 免密码登录。
首先配置每一个节点的 hosts 文件：

```
[root@hadoop-cm0 ~]# cat /etc/hosts
127.0.0.1 localhost.localdomain localhost
192.168.0.150 hadoop-cm0
192.168.0.151 hadoop-cm1
192.168.0.152 hadoop-cm2
192.168.0.153 hadoop-cs1
192.168.0.154 hadoop-cs2
192.168.0.155 hadoop-cs3
```

在每一个节点执行：

```
[root@hadoop-cm0 ~]# ssh-keygen -t  rsa
```

然后将本节点生成的公钥拷贝至其他节点：

```
[root@hadoop-cm0 ~]#ssh-copy-id 192.168.1.150
[root@hadoop-cm0 ~]#ssh-copy-id 192.168.1.151
…
```

拷贝过程中需要根据提示输入 root 密码。拷贝完成后，可以使用如下命令来测试 ssh 免密码登录配置是否成功：

```
[root@hadoop-cm0 ~]# ssh hadoop-cm0
Last login: Fri Nov 14 15:06:59 2014 from 192.168.0.100
[root@hadoop-cm0 ~]# exit
logout
Connection to hadoop-cm0 closed.
[root@hadoop-cm0 ~]# ssh hadoop-cm1
Last login: Fri Nov 14 15:56:58 2014 from 192.168.0.100
[root@hadoop-cm1 ~]#
…
```

在每次执行该命令后相当于登录到了另外一个节点上，需要执行一个退出动作，再进行下一个节点的测试。

11. 配置时间同步

一般在企业内网中都有单独的 NTP 时间同步服务器，我们需要做的就是修改配置文件，加入企业自己的时间同步服务器 IP 地址（/etc/ntp.conf）。

然后启用 ntp 服务：

```
[root@hadoop-cm0 ~]#service ntpd start
```

12. 配置内核参数

（1）参数 1：kernel.mm.redhat_transparent_hugepage.defrag

该参数默认值为 always，这可能带来 CPU 利用率过高的问题，需将其设置为 never。

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag
```

为了保证重启生效，我们可以把该命令写到 rc.local 中，保证每次开机都能调用。

```
echo 'echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag' >>
/etc/rc.local
```

（2）参数 2：vm.swappiness

该参数默认值为 60，这里将其设置为 0，让操作系统尽可能不使用交换分区，有助于提高集群的性能。将该参数写入配置文件 sysctl.conf：

```
echo "vm.swappiness = 0" >> /etc/sysctl.conf
```

运行如下命令使修改生效：

```
sysctl -p
```

13. 配置图形界面

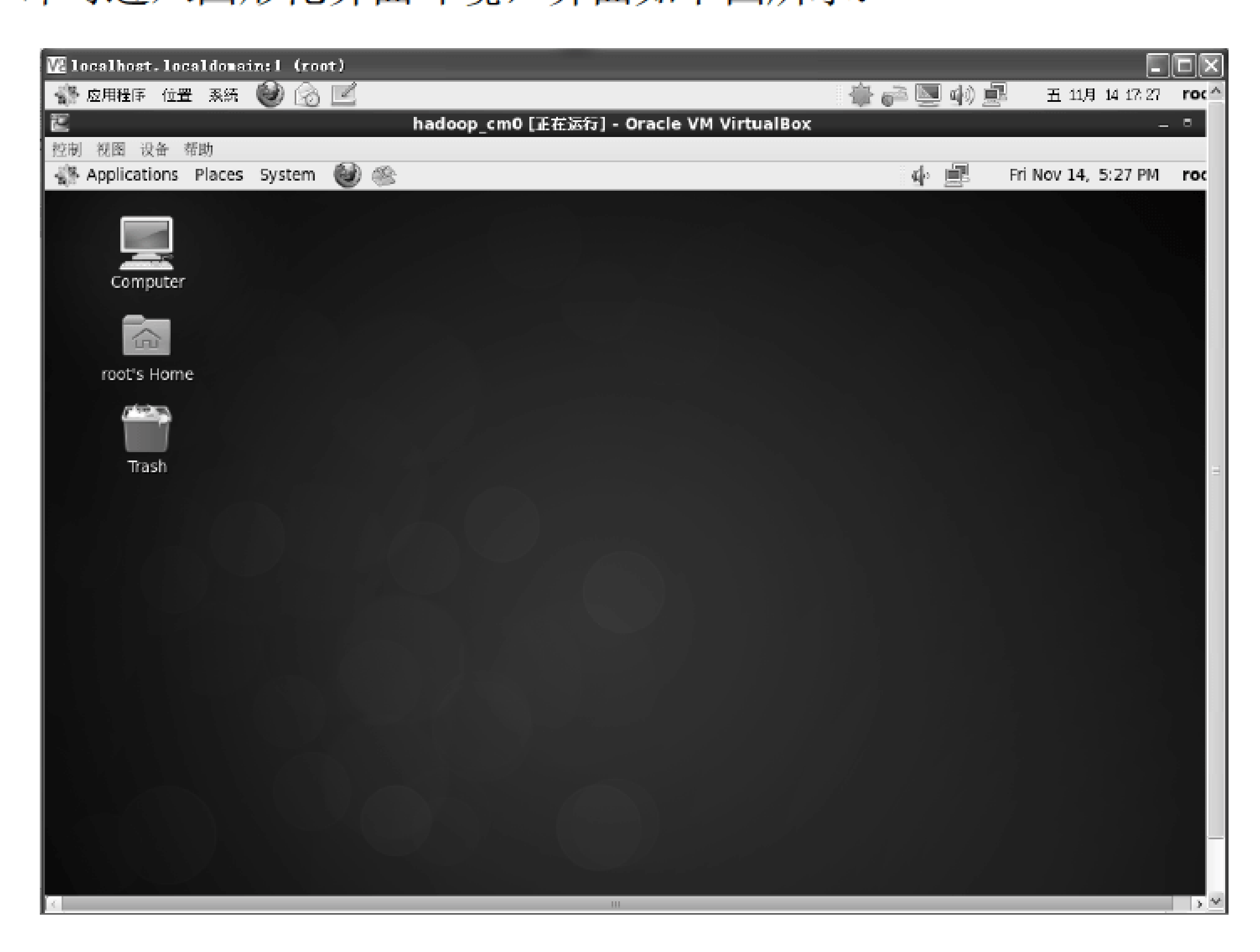
CM 安装初始阶段需要图形化的界面环境，我们可以使用 Xmanger 或者 VNC。本示例中仅在 hadoop-cm0 上启动 vncserver（vncserver 的配置本处不再赘述）。

```
[root@hadoop-cm0 ~]# vncserver
New 'hadoop-cm0:1 (root)' desktop is hadoop-cm0:1
Starting applications specified in /root/.vnc/xstartup
Log file is /root/.vnc/hadoop-cm0:1.log
```

然后我们通过 VNC 客户端工具就可以连接过去了，界面如下图所示：



输入密码，即可进入图形化界面环境，界面如下图所示：



14. 下载 cloudera-manager-installer.bin

该文件是整个安装的入口，可以从如下位置下载：

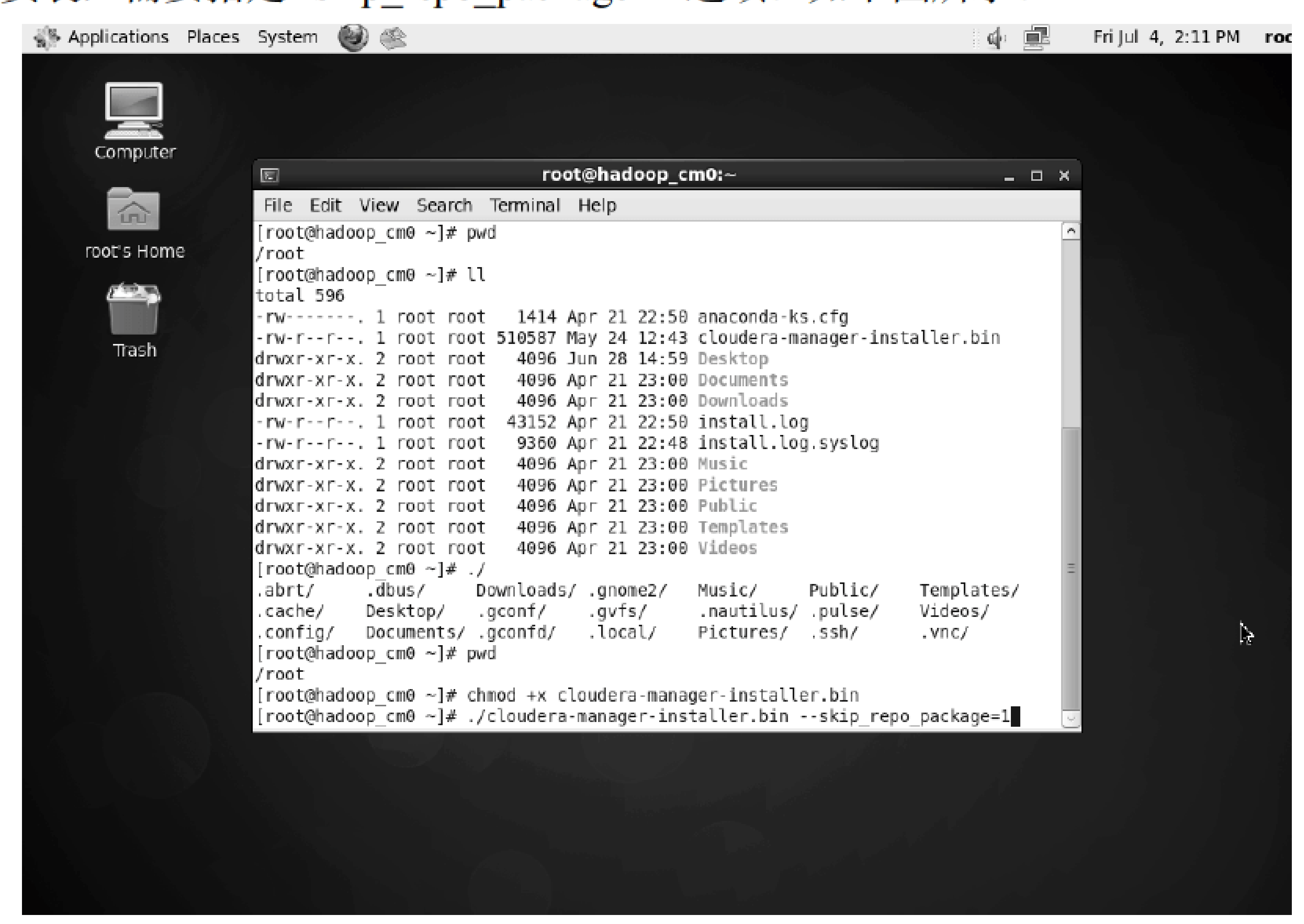
```
http://archive-primary.cloudera.com/cm5/installer/5.0.2/
```

下载后需传到 CM 服务器本机，本示例中为 hadoop-cm0。

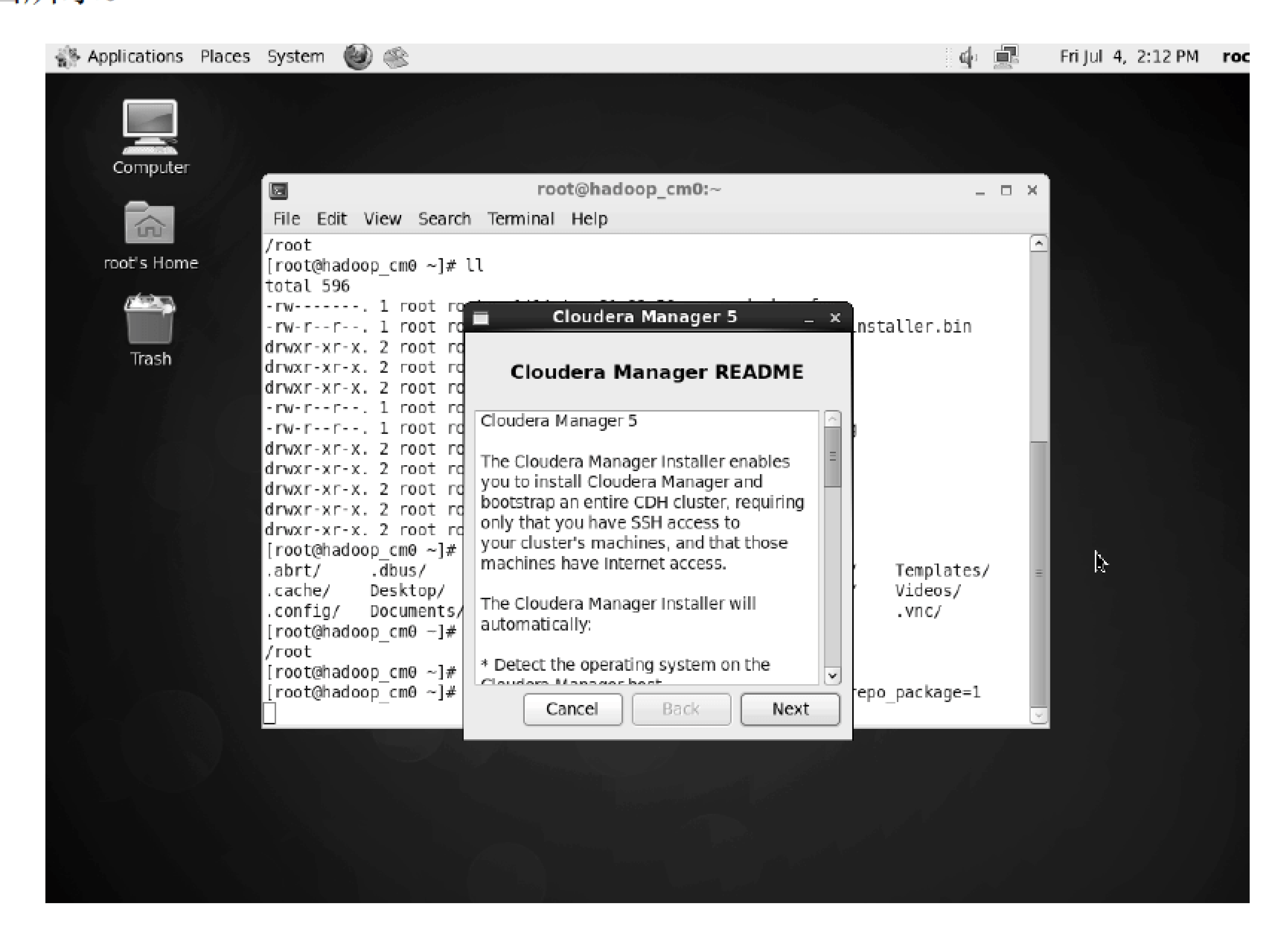
1.3 CM 及 CDH 安装

1. 启动 CM 安装

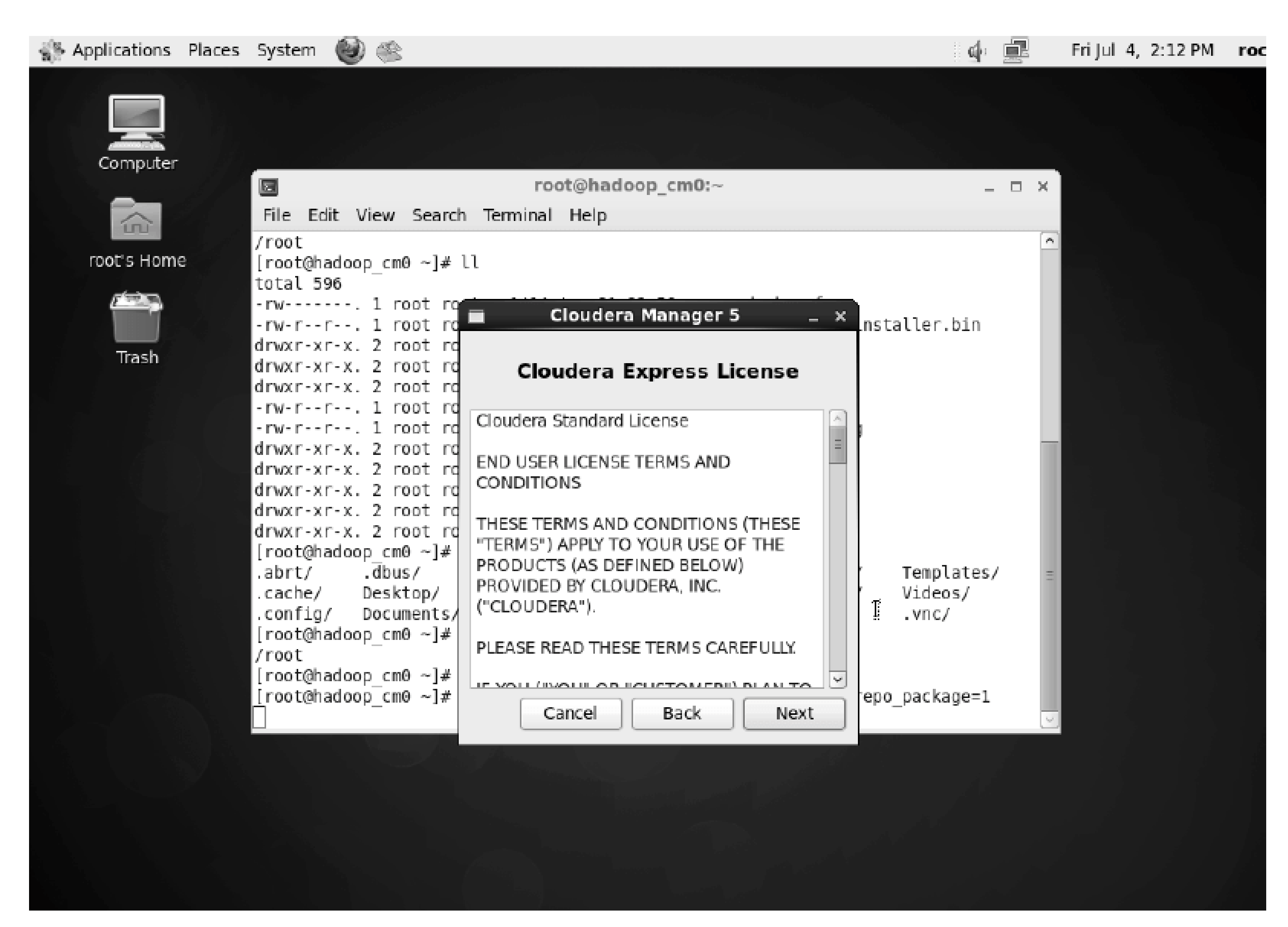
首先需要给文件 cloudera-manager-installer.bin 赋予可执行权限。另外，如果是像本示例一样进行的是离线安装，需要指定--skip_repo_package=1 选项。如下图所示。



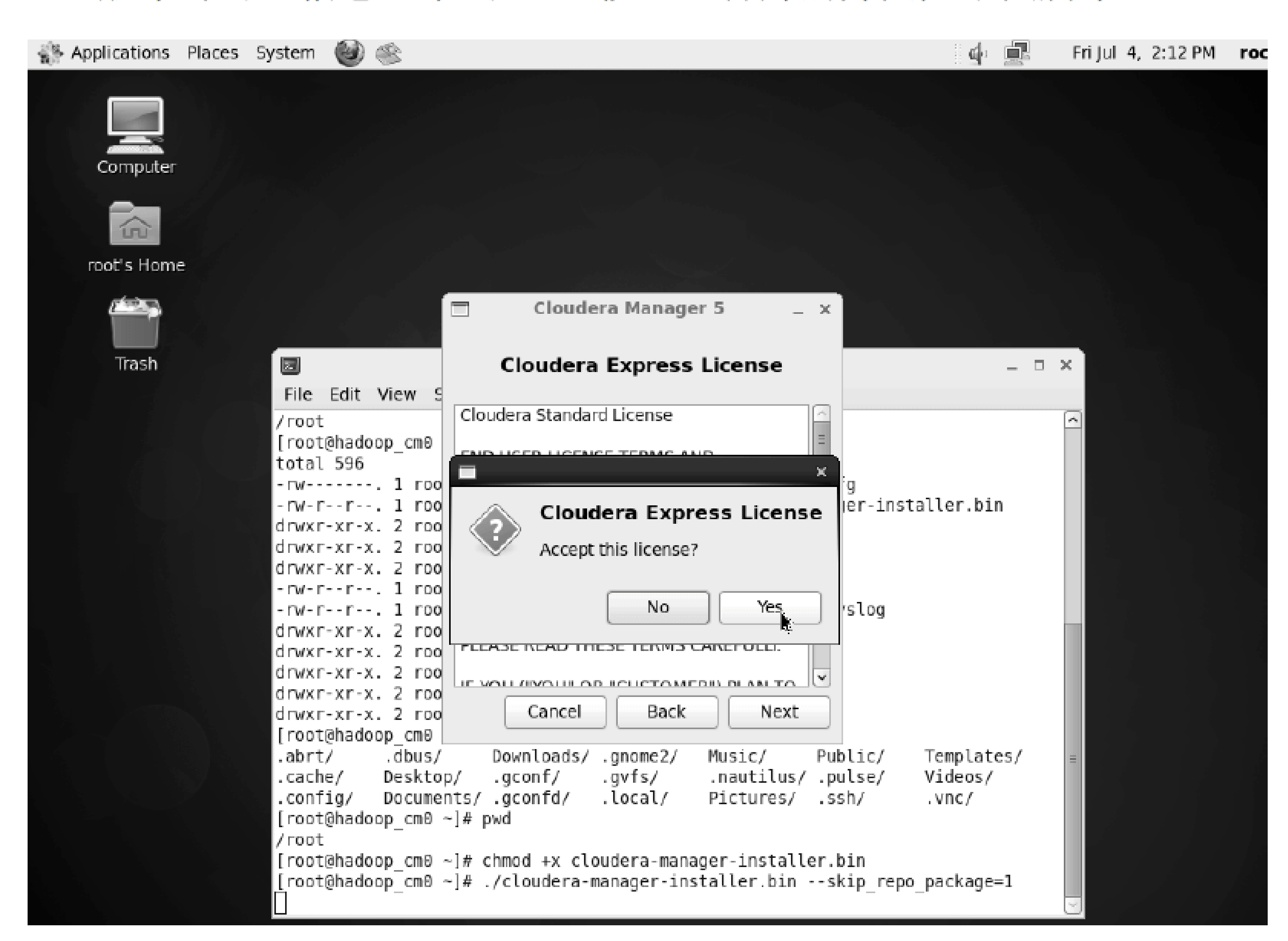
在终端上运行命令 cloudera-manager-installer.bin --skip_repo_package=1，开始安装 CM，打开界面如下图所示。



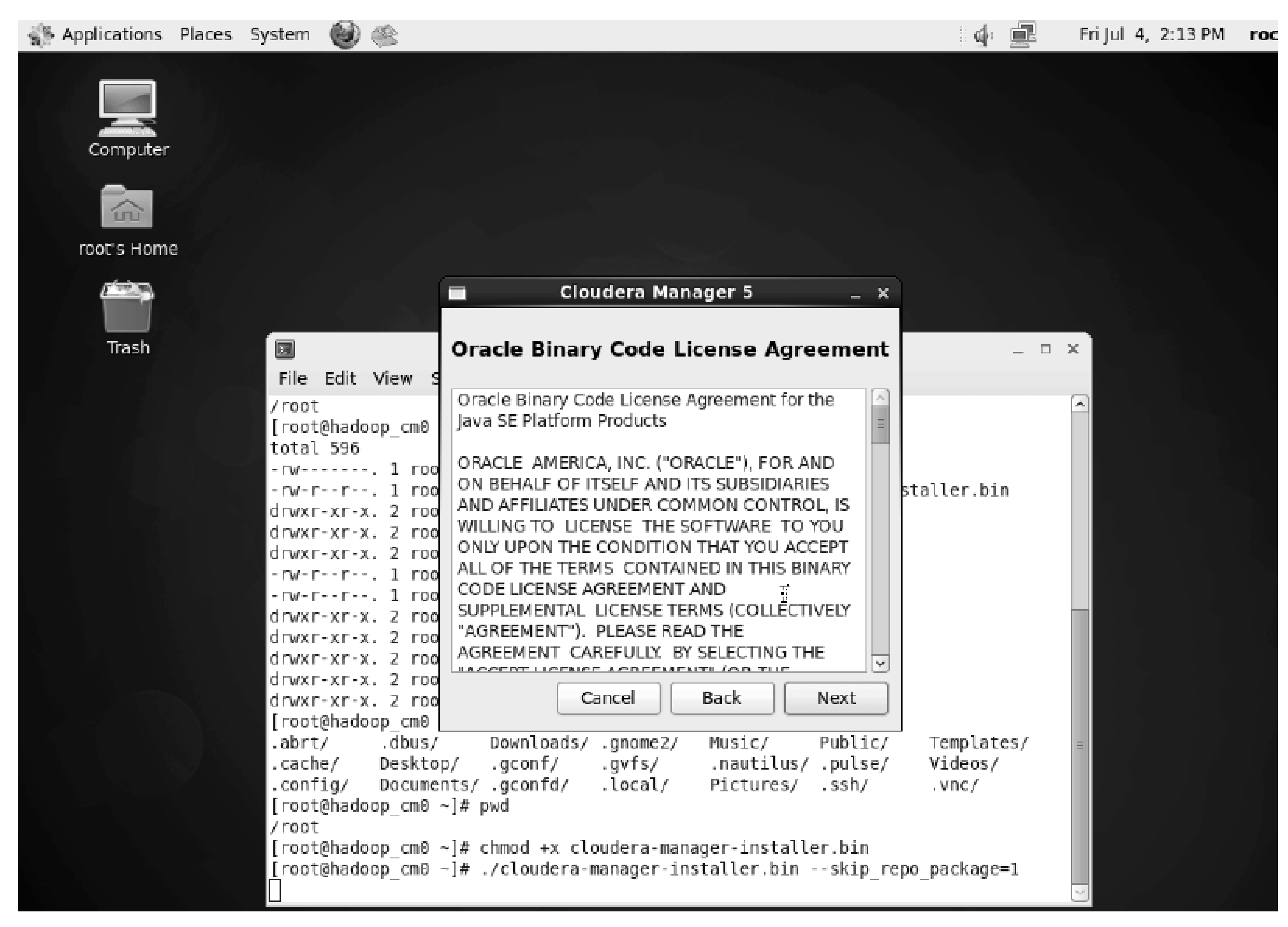
该部分提示是安装概述，提示我们要保持各节点连接畅通，并且仅通过 SSH 在各节点间通信。单击 Next 按钮，打开的界面如下图所示。



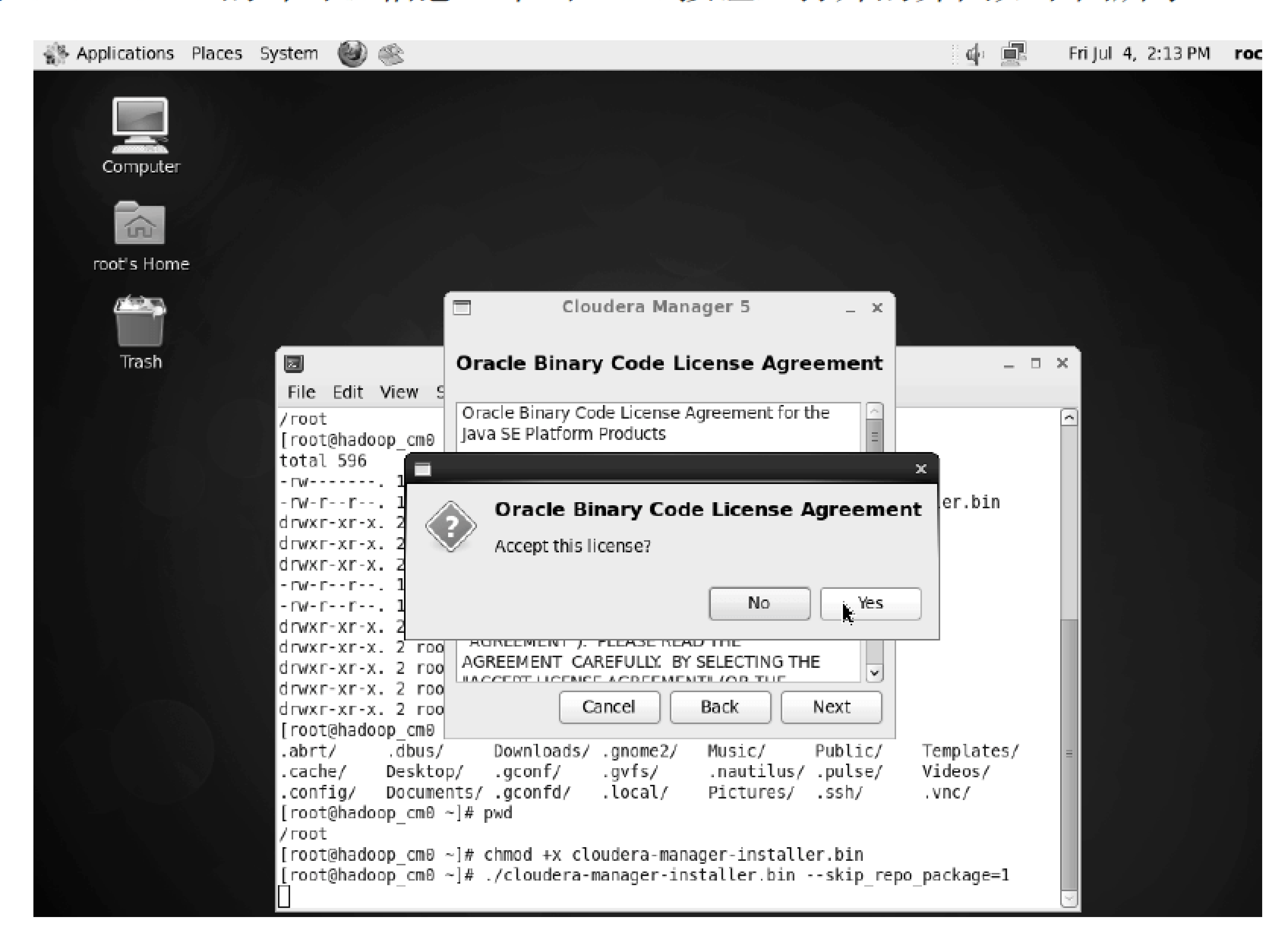
此处为 CM 版本许可证信息。单击 Next 按钮，打开的界面如下图所示。



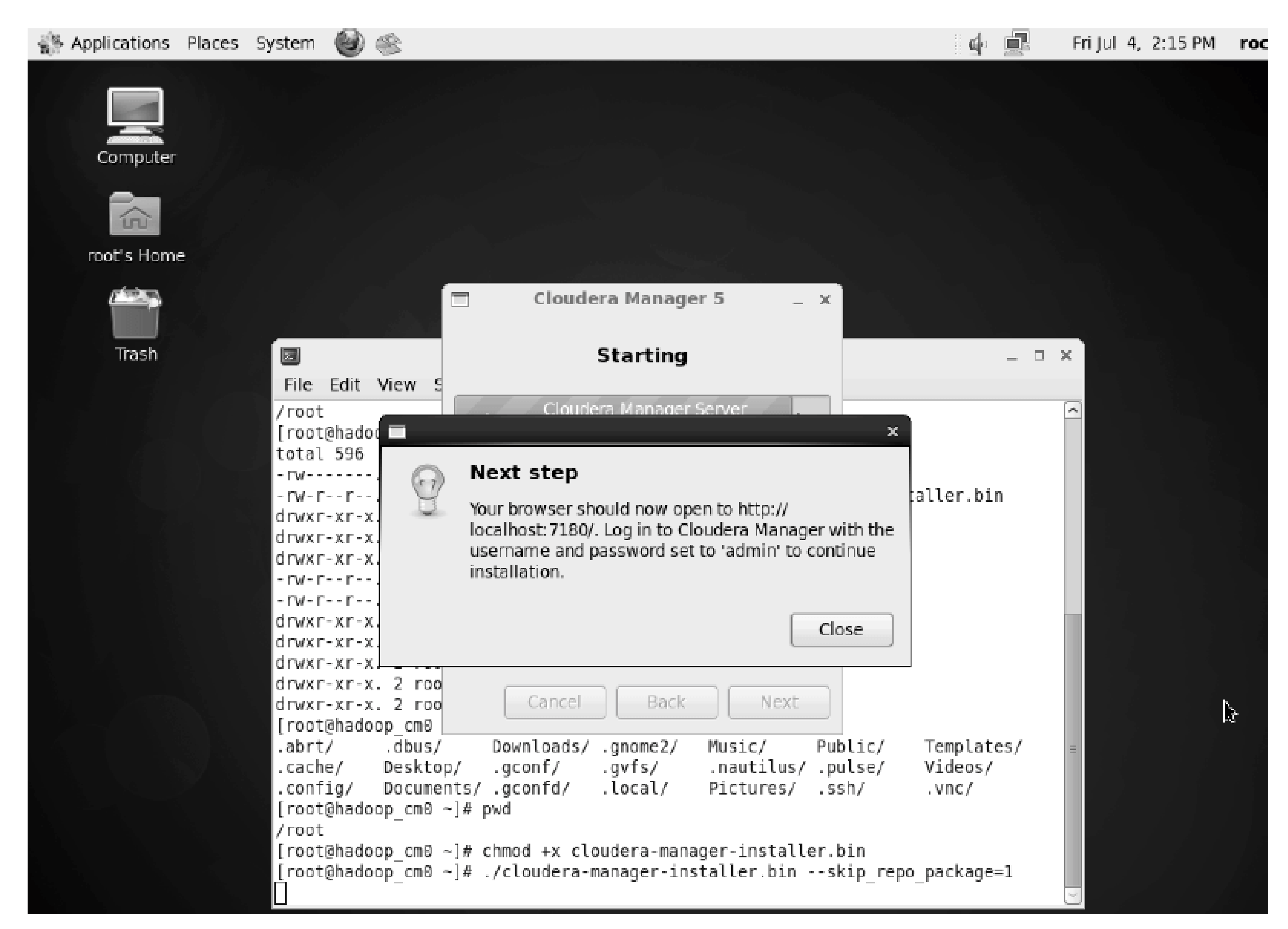
此处提示我们接受该许可证，必须选 Yes，打开的界面如下图所示。



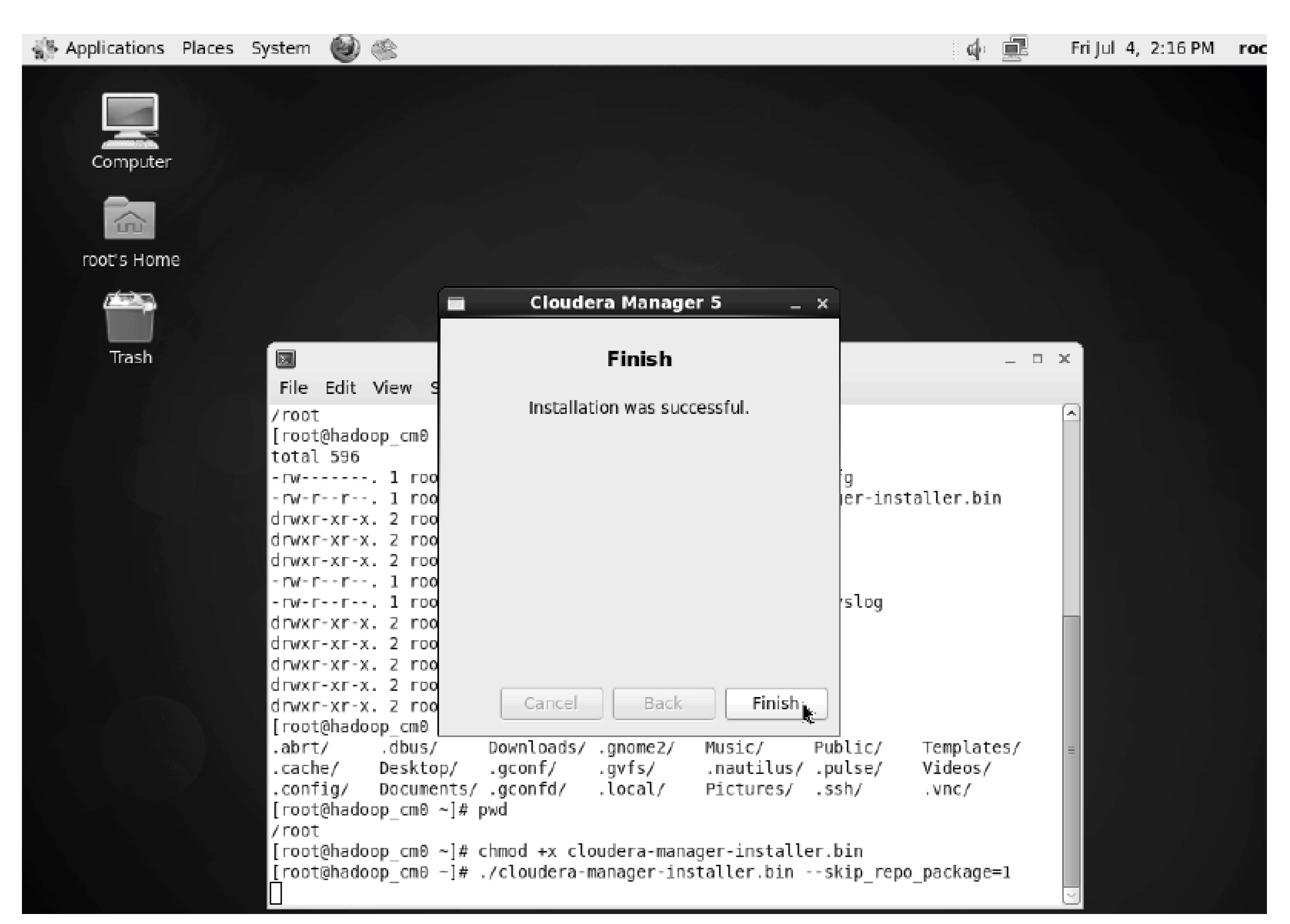
此处为 Oracle JDK 的许可证信息。单击 Next 按钮，打开的界面如下图所示。



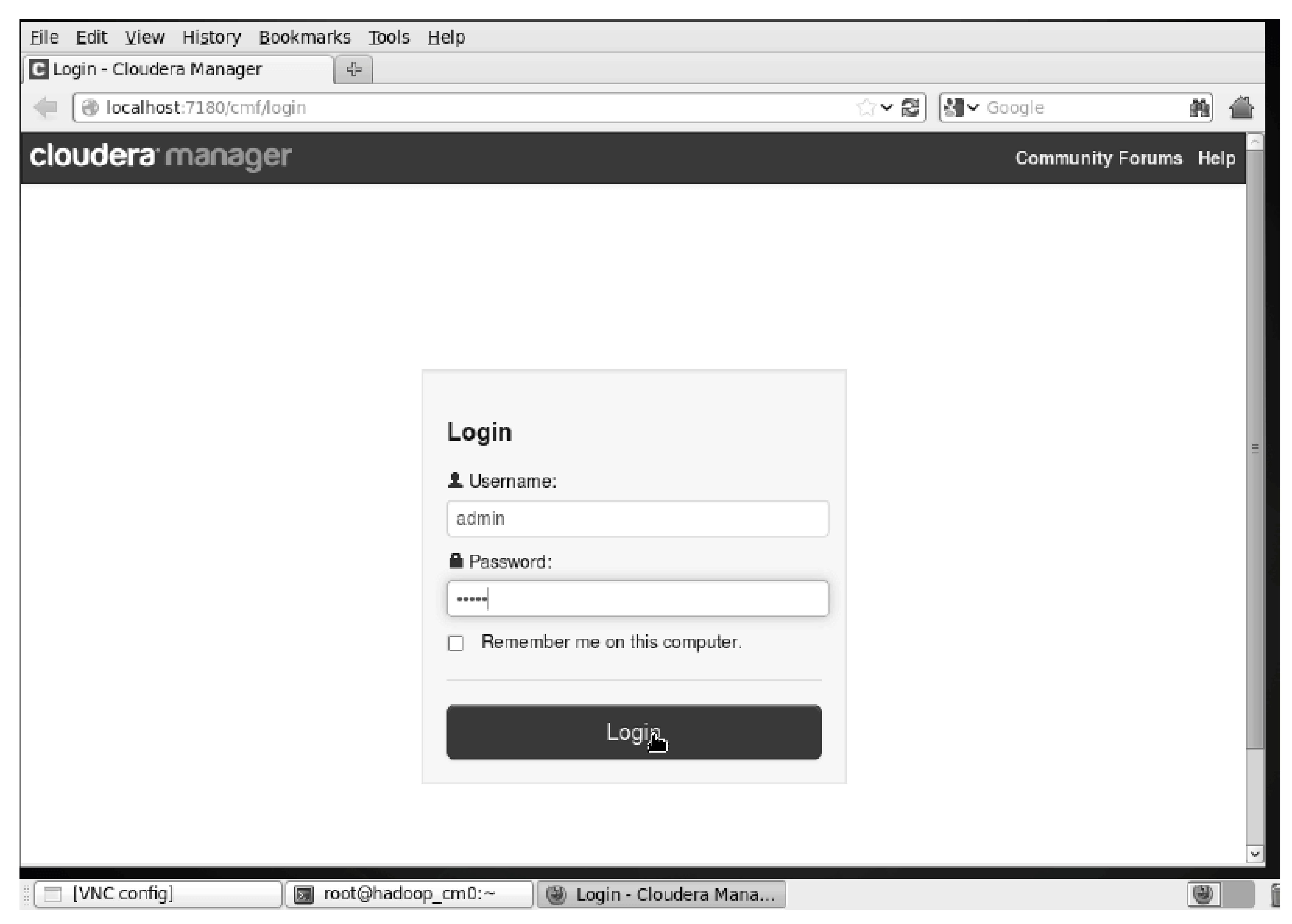
此处必须接受该许可证，选择 Yes 按钮。随后启动如下安装过程：



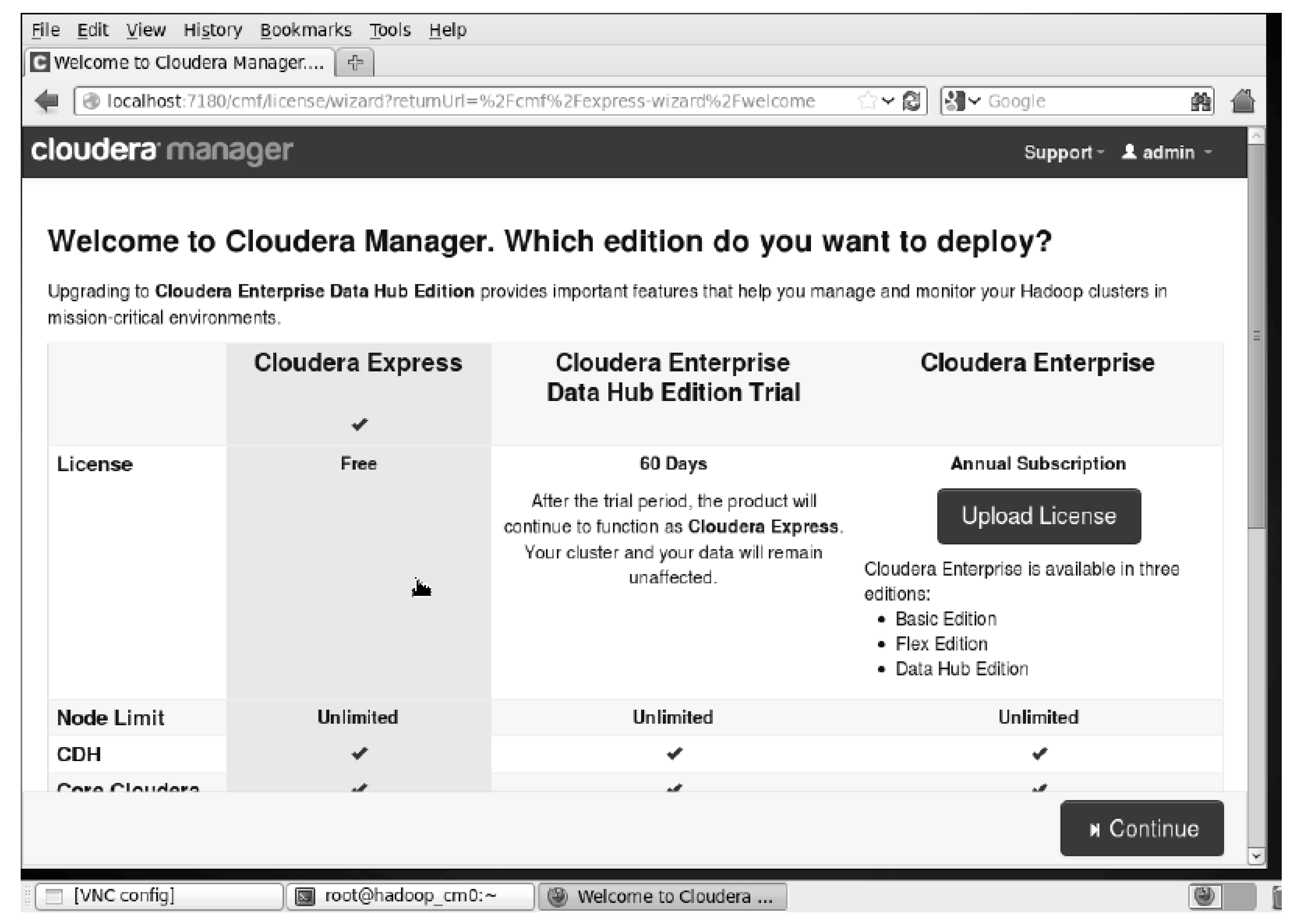
安装快结束时，提示我们使用用户名 admin 密码 admin 访问浏览器 http://localhost:7180。这就是未来 CM 管理界面的入口。打开的界面如下图所示。



单击 Finish 按钮，安装完成。打开浏览器，在地址栏中输入提示我们输入的 url：http://localhost:7180，用户名和密码，界面如下图所示。

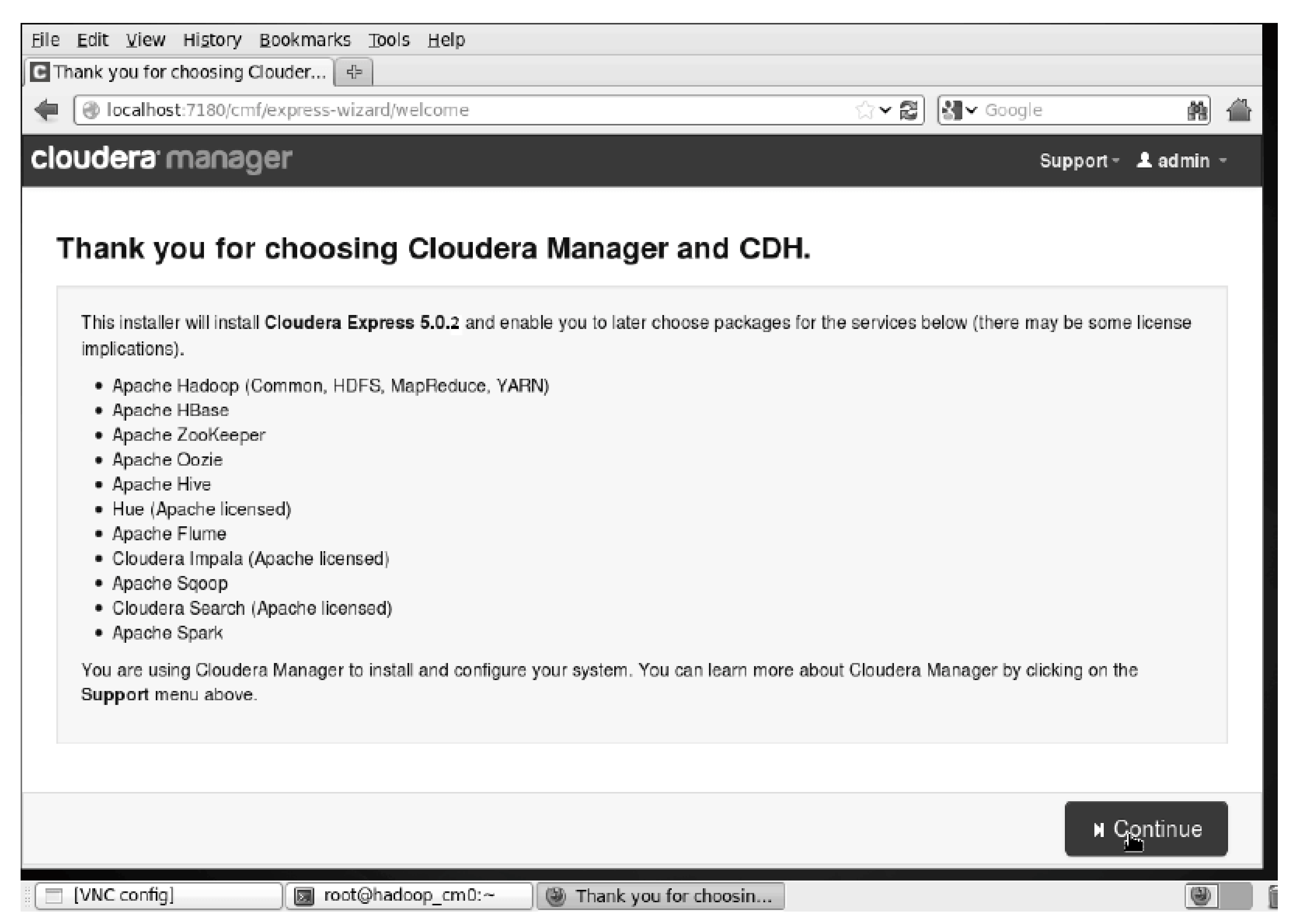


单击 Login 按钮，登录系统，打开的界面如下图所示。

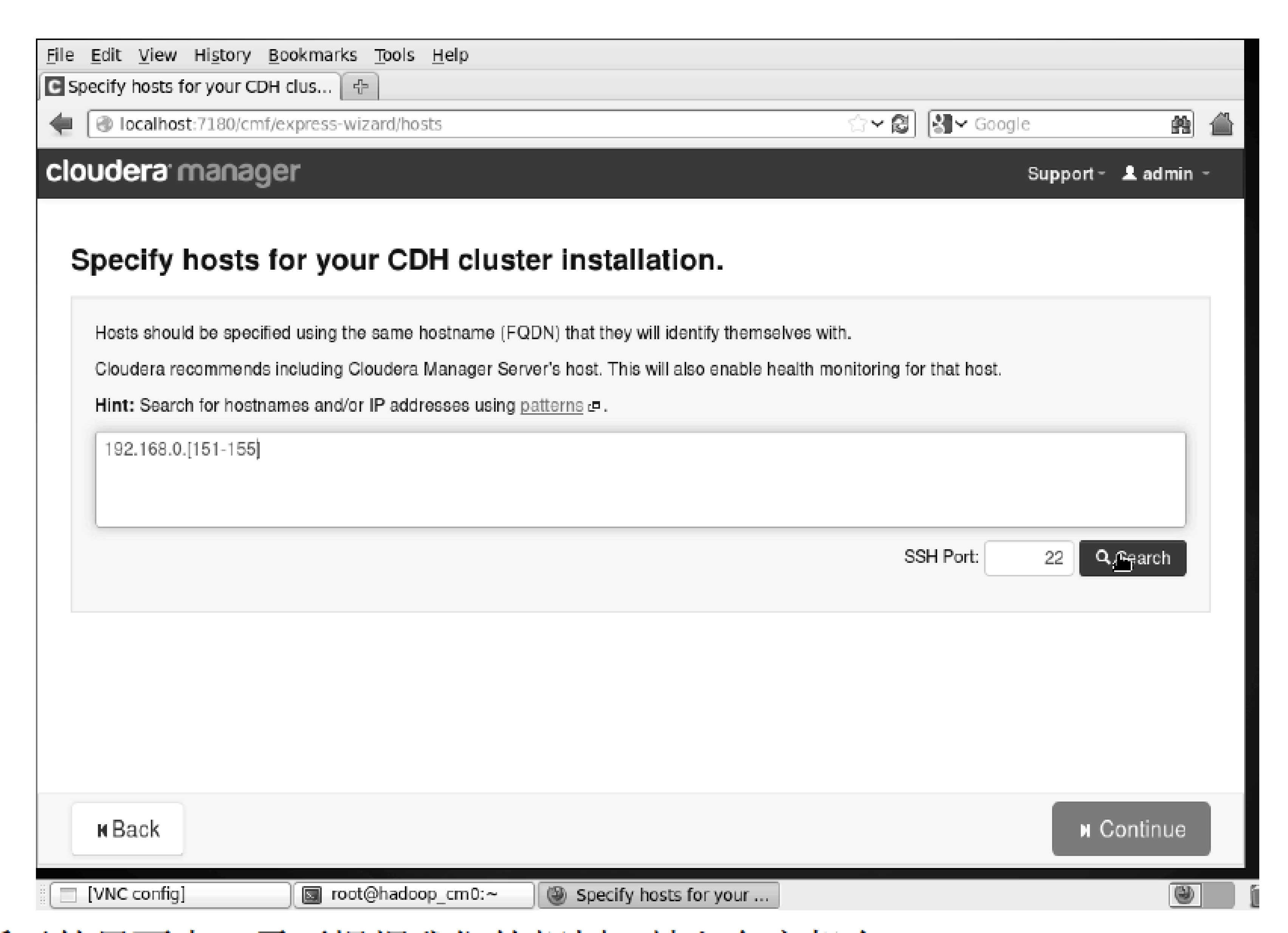


这里我们可以看到 CM 有三个不同的版本，Enterprise 版有额外的功能，是收费的。Express 版是免费的，功能相对较少，在较早的版本有安装的节点数限制，在 CM5.0.2 中已经没有节点数限制了。Data Hub 版提供有 Enterprise 版的 60 天试用期，试用期过后，自动转换成 Express 版。

此处我们选择 Express 版，单击 Continue 按钮，打开的界面如下图所示。



本部分提示我们可以使用 CM 来部署和管理的组件。单击 Continue 按钮，打开的界面如下图所示。

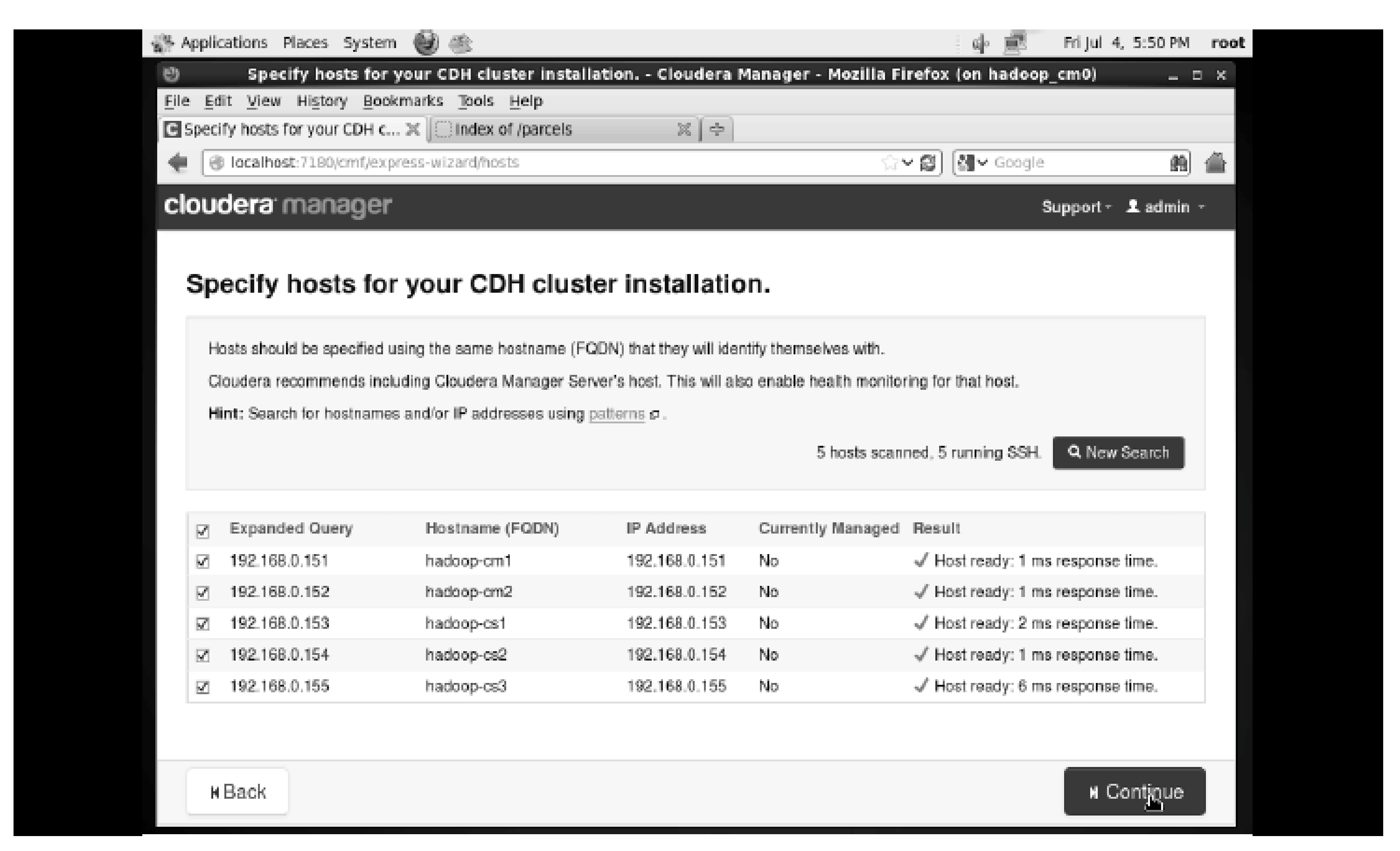


上图所示的界面中，需要根据我们的规划，填入各主机名。

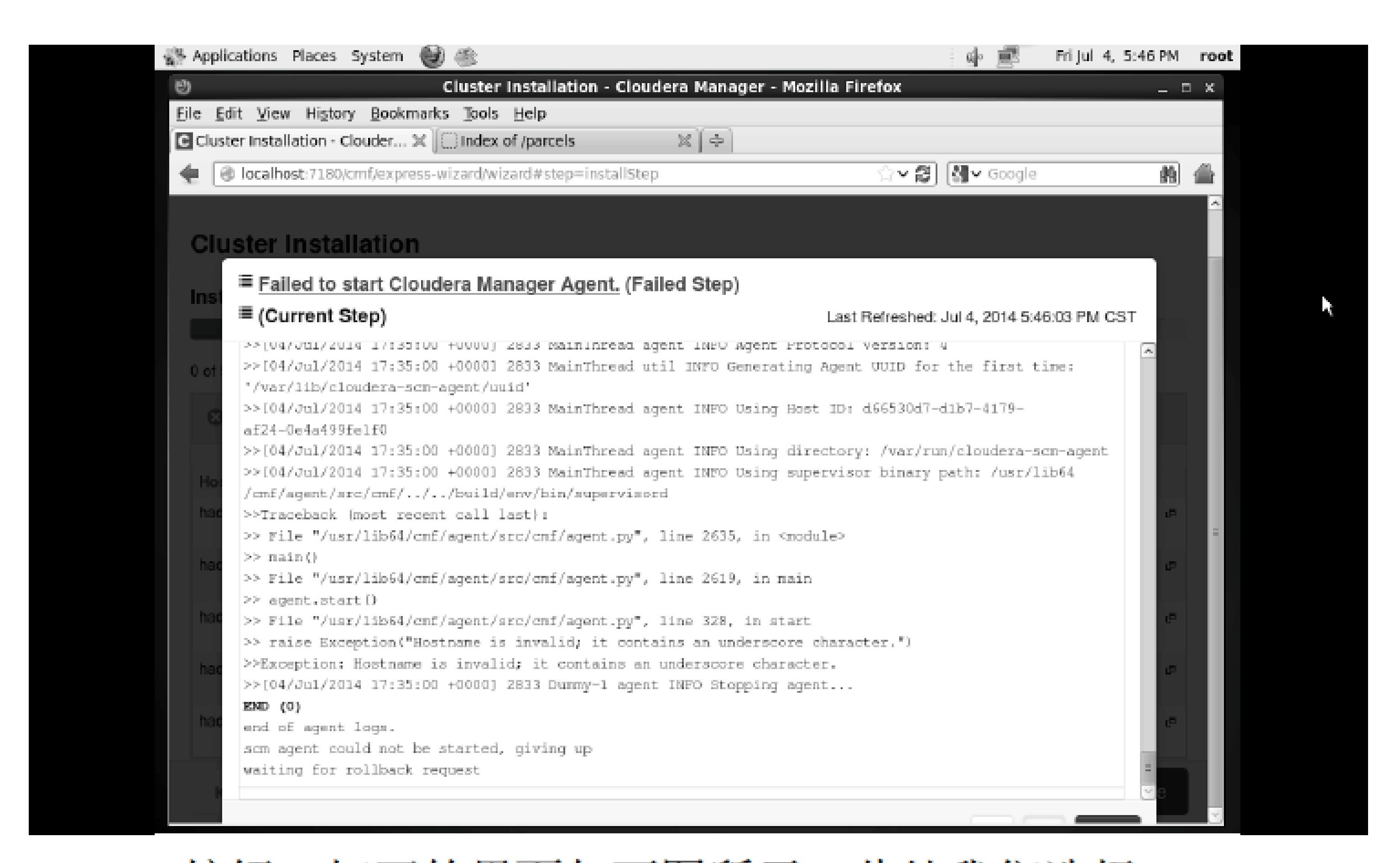
需要注意的是，我们可以使用类似正则表达式的方式来表示主机名或者 IP 地址，对于动辄上百上千个节点的集群来说，给我们提供了很大的便利。

这里，我们需要安装的 IP 是从 192.168.0.151 到 192.168.0.155，可以使用 192.168.0.[151-155] 来表示。

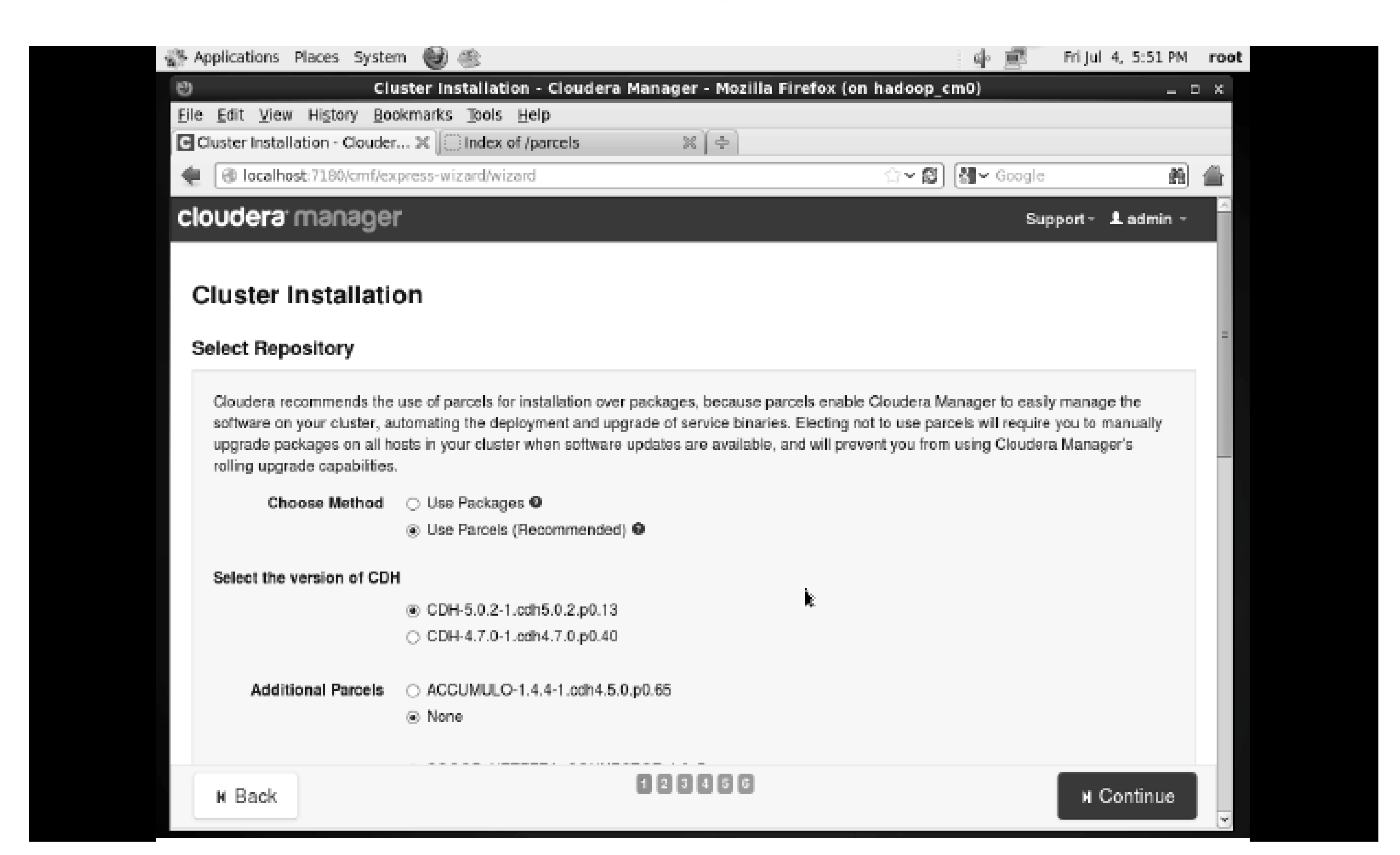
为了确认我们填入的主机名或者 IP 地址是否正确，可以让 CM 自动搜索，单击 Search 按钮。搜索之后，可以看到我们刚才输入的各节点对应的主机名以及连接的响应时间等，界面如下图所示。



注意：在安装过程中主机名不得有下划线，如果有下划线，此处将会报如下图所示的错误：



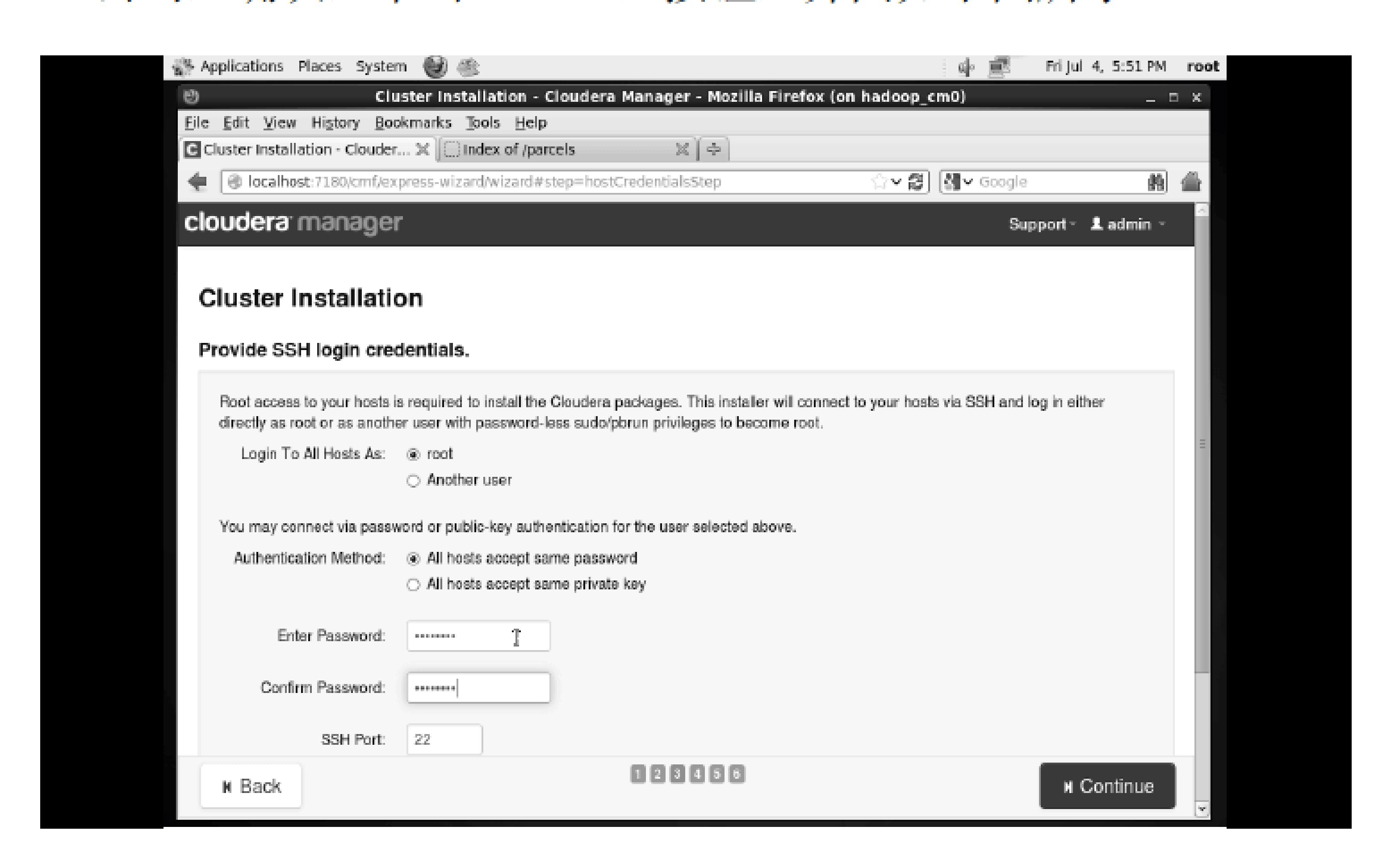
继续单击 Continue 按钮，打开的界面如下图所示。此处我们选择 Use Parcels 选项。



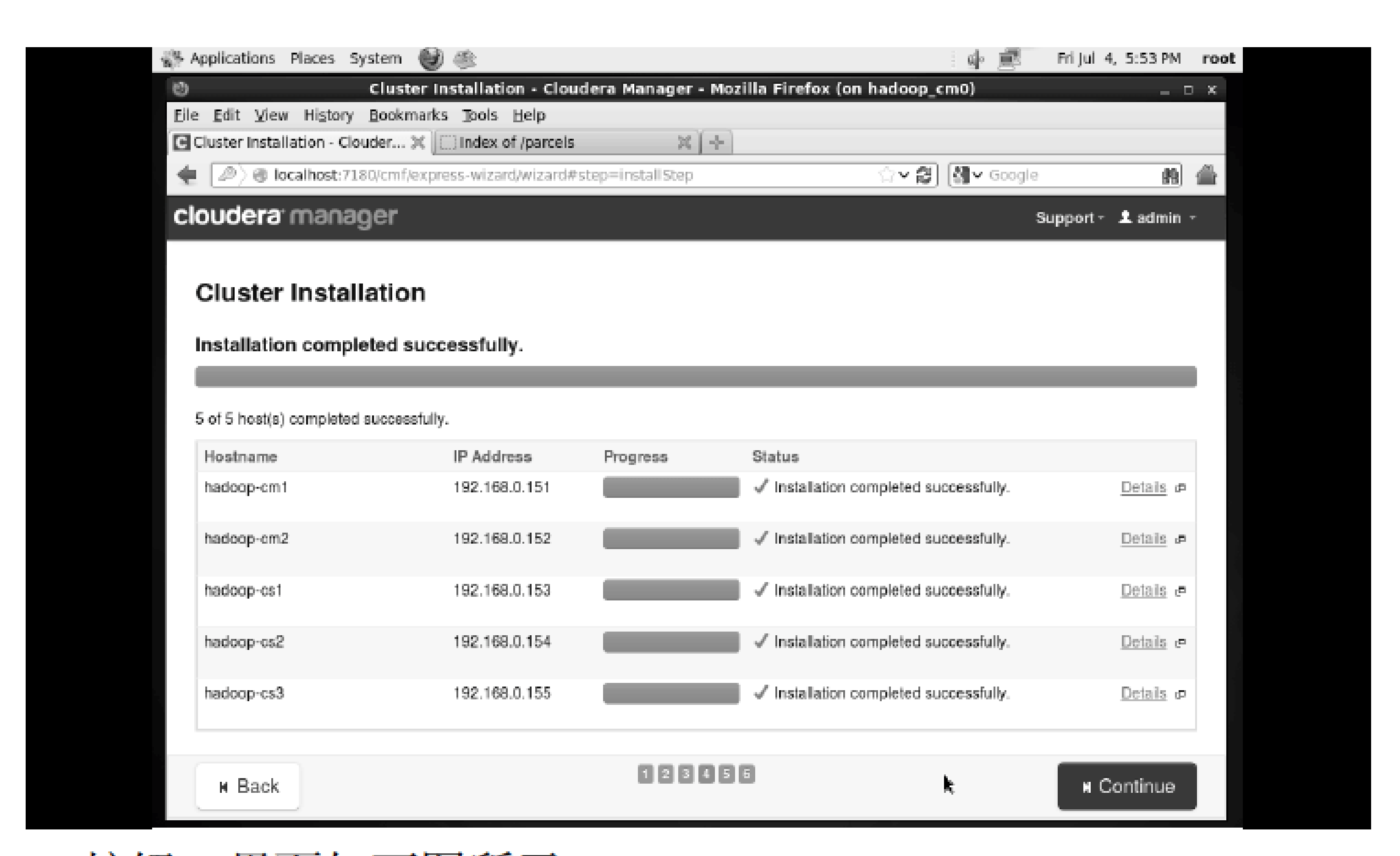
在上图所示的界面上，分别指定 CM 和 CDH 的本地 yum 源地址，本示例中分别为：

```
http://192.168.0.10/cm502
http://192.168.0.10/parcels
```

此处输入 root 密码，确认，单击 Continue 按钮，界面如下图所示。



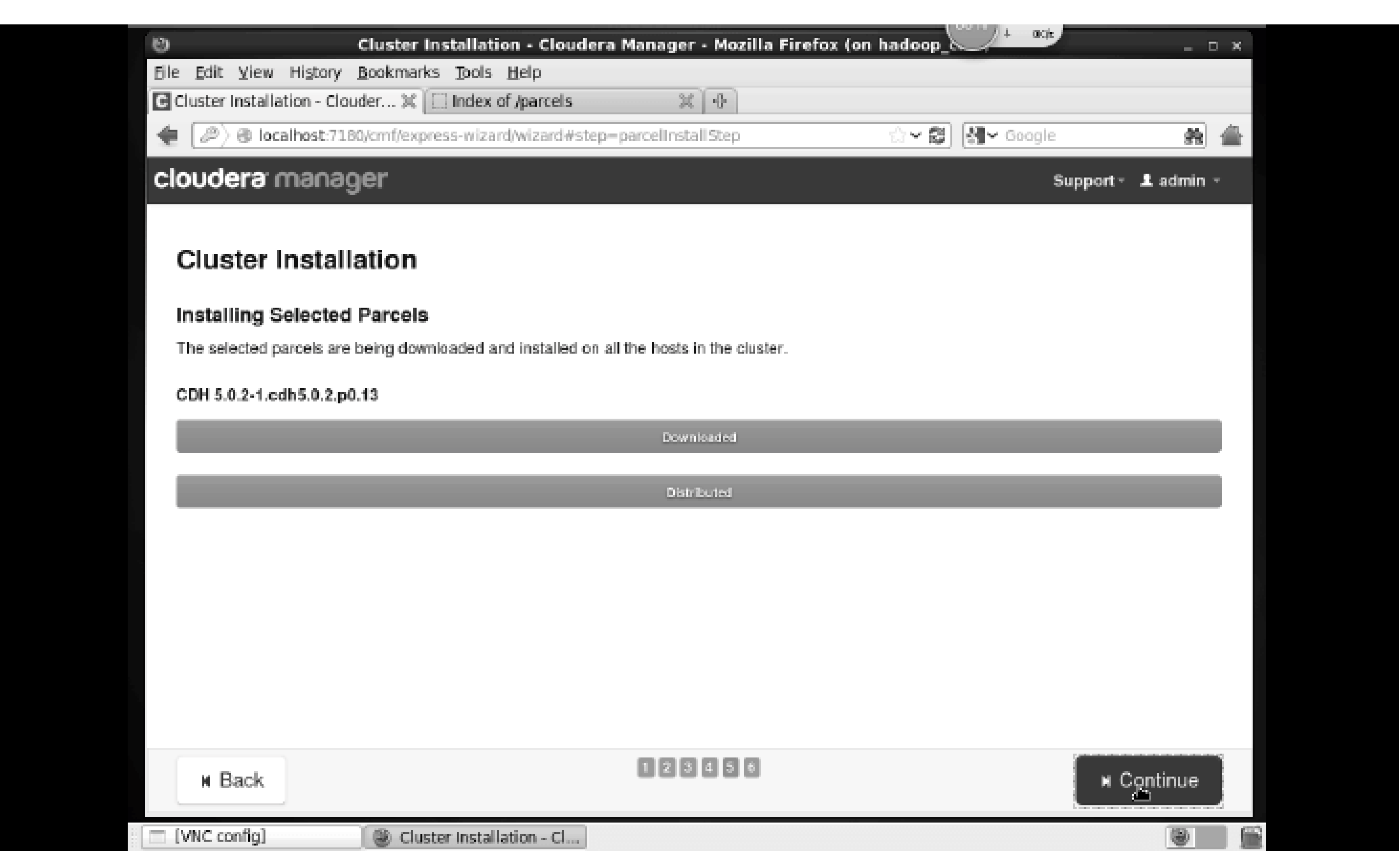
CM 会先安装 JDK 及其他一些必须的安装包，安装完成后，界面如下图所示。



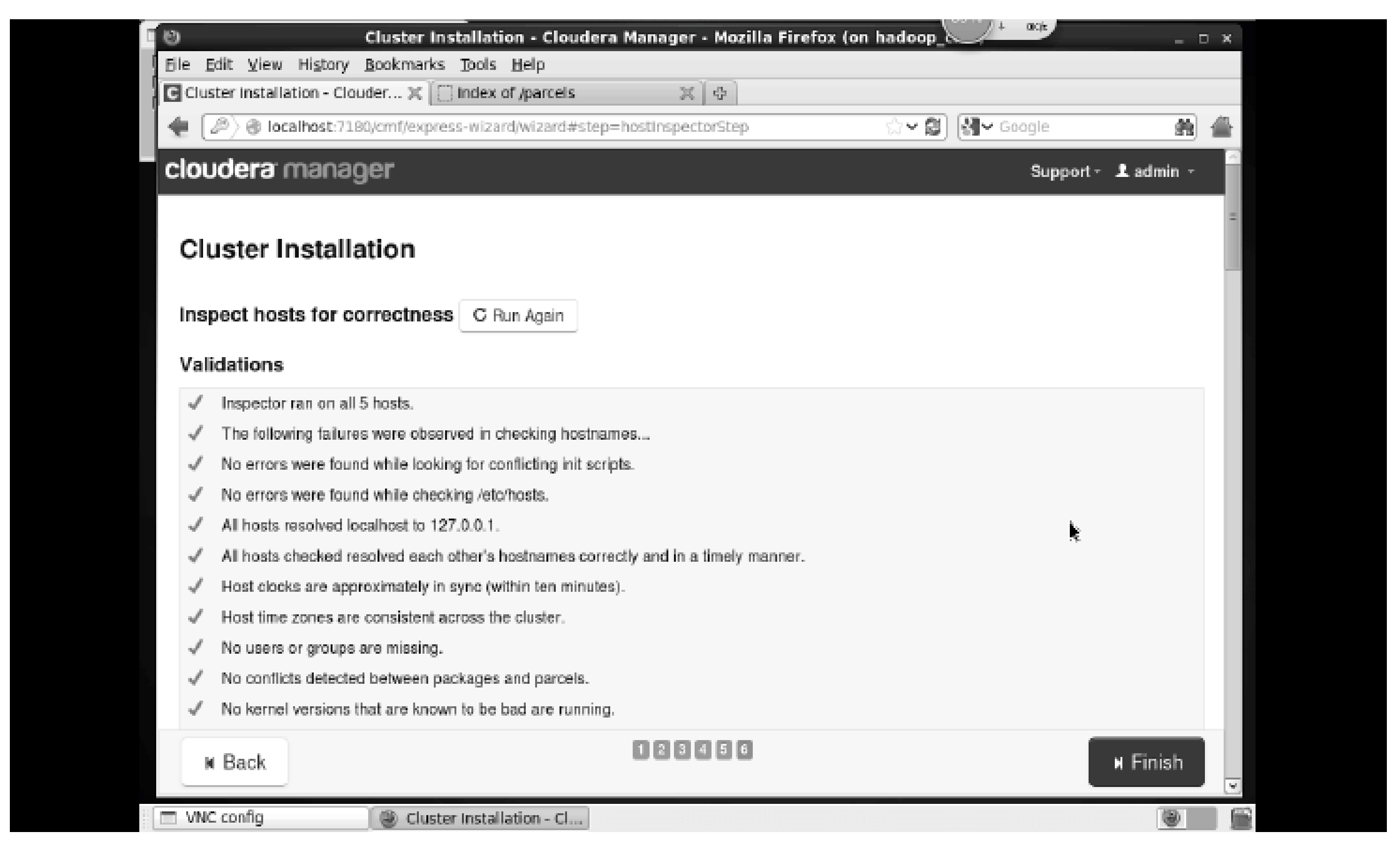
单击 Continue 按钮，界面如下图所示。



此处开始进入 CDH 安装过程，CM 负责将 parcel 文件分发到各节点。单击 Continue 按钮，界面如下图所示。

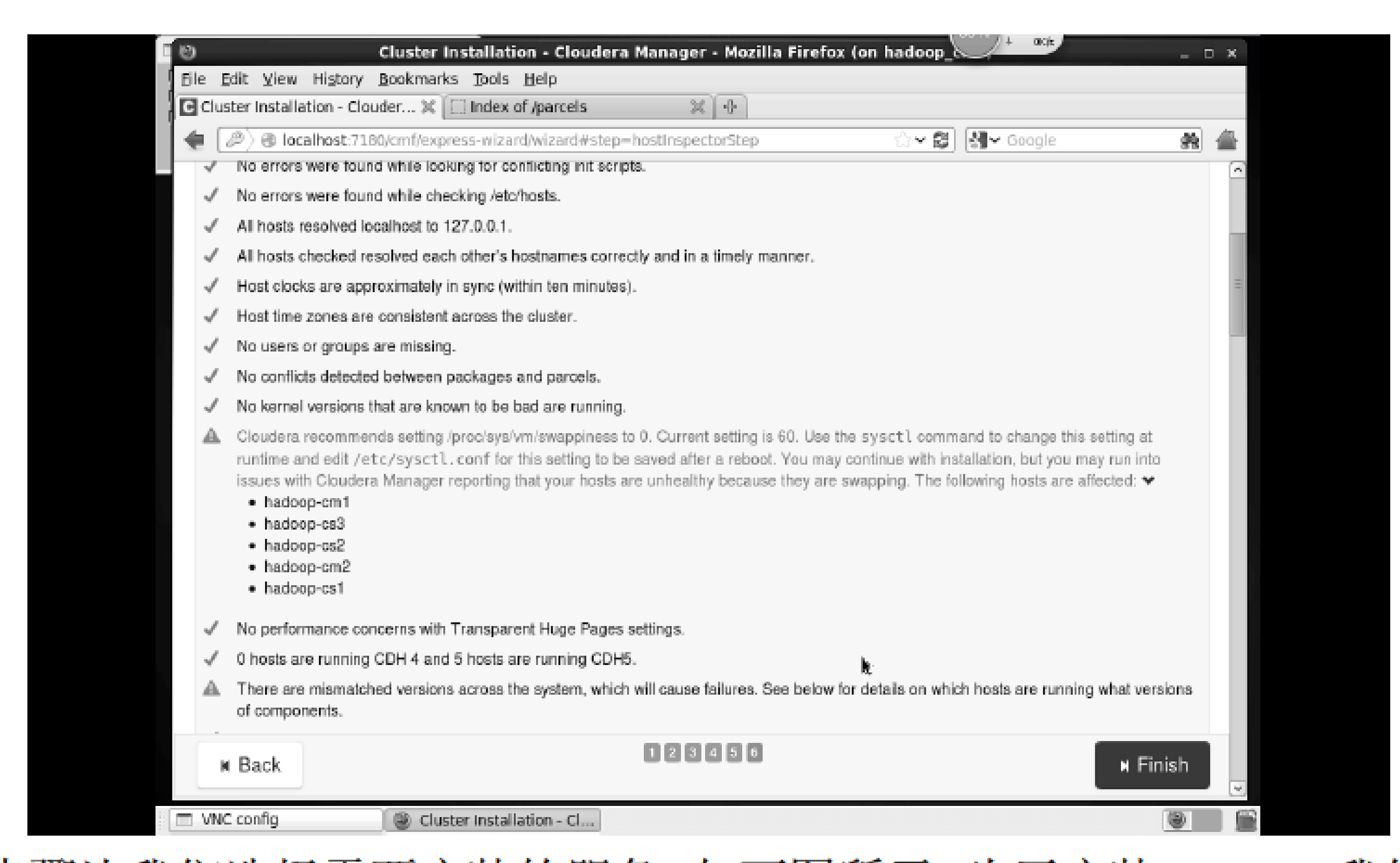


parcel 包分发完成。单击 Continue 按钮，界面如下图所示。



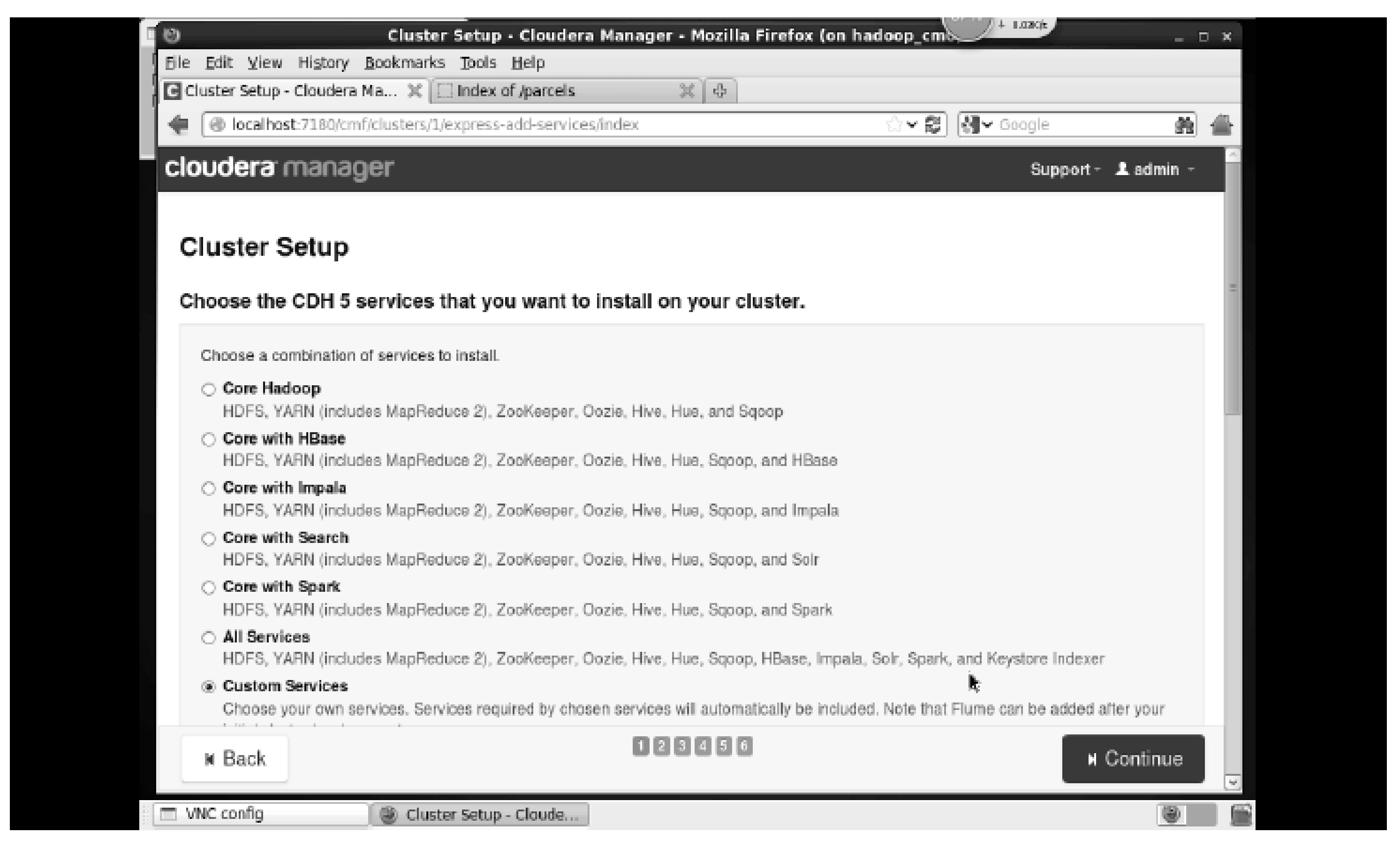
此处单击 Inspector 会进行安装前各节点的检查工作，单击一次之后变为 Run Again。单击 Finish 按钮，界面如下图所示。

注意：如果遇到如下图所示的问题，请参照第 1.2 节 CM 安装准备第 10 步执行。



接下来的步骤让我们选择需要安装的服务，如下图所示。为了安装 Impala，我们可以选择 Core

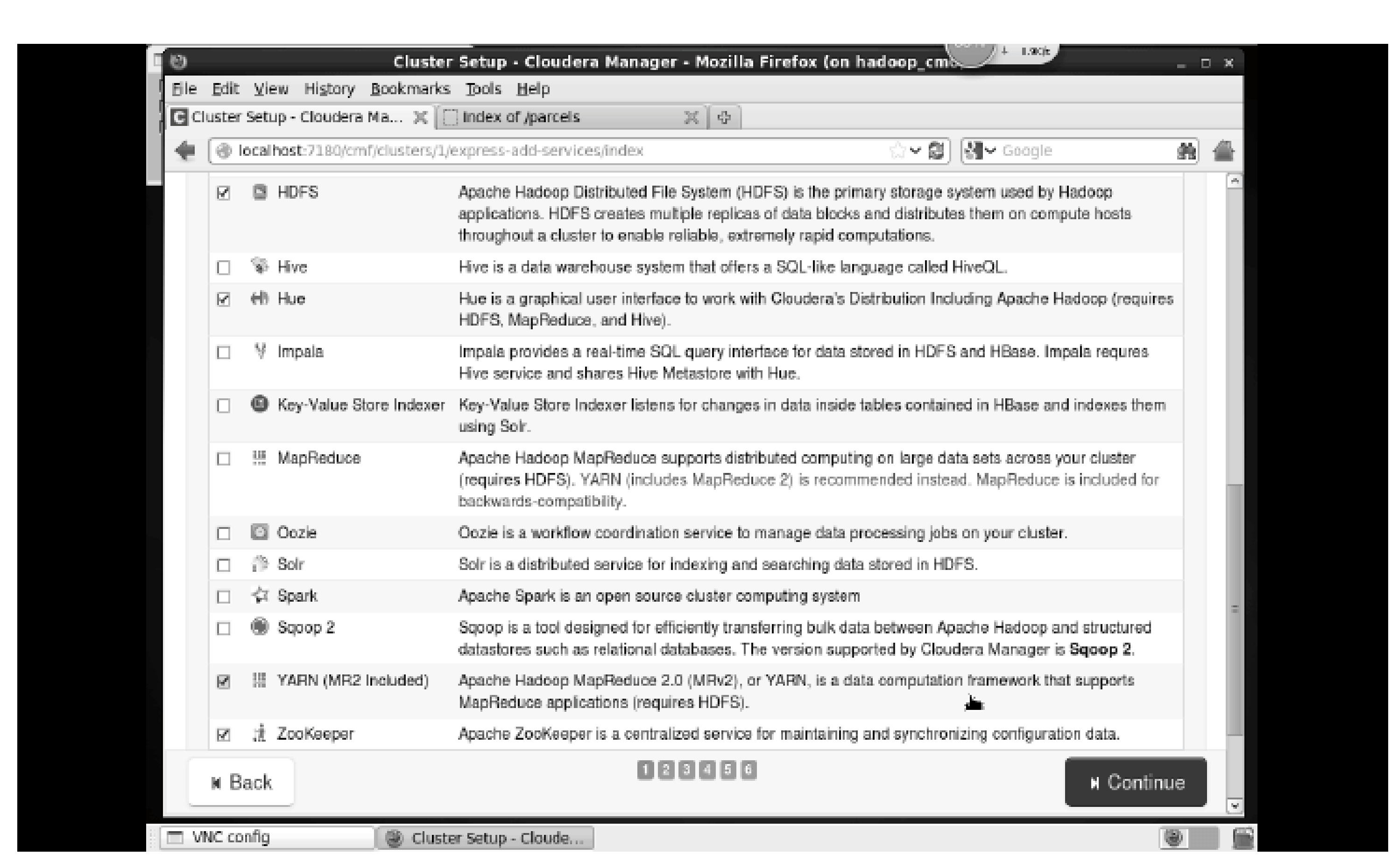
With Impala 选项。



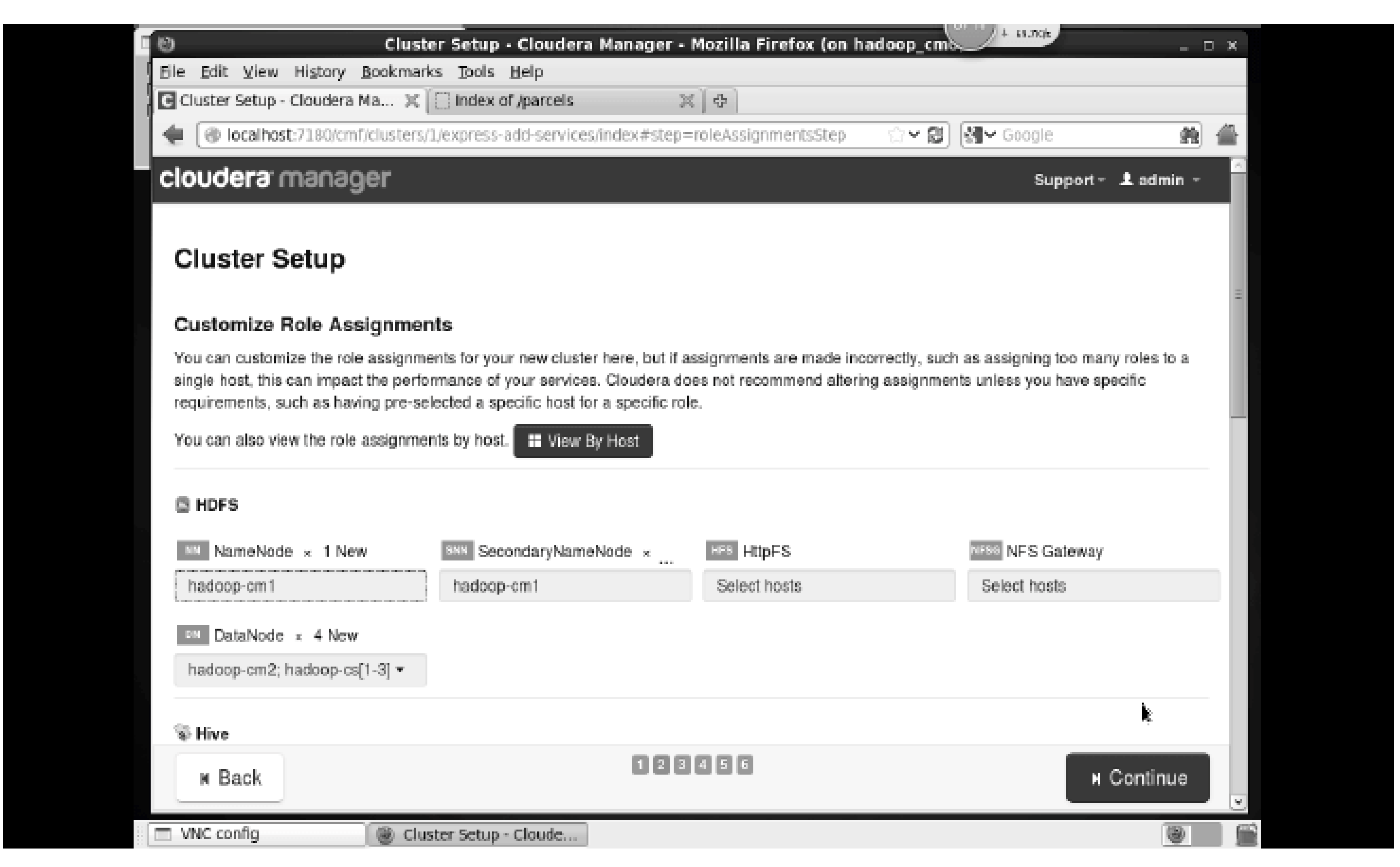
但是如果选择该选项，将会启动如下服务：

- HDFS
- YARN
- Zookeeper
- Oozie
- Hive
- Hue
- Sqoop
- Impala

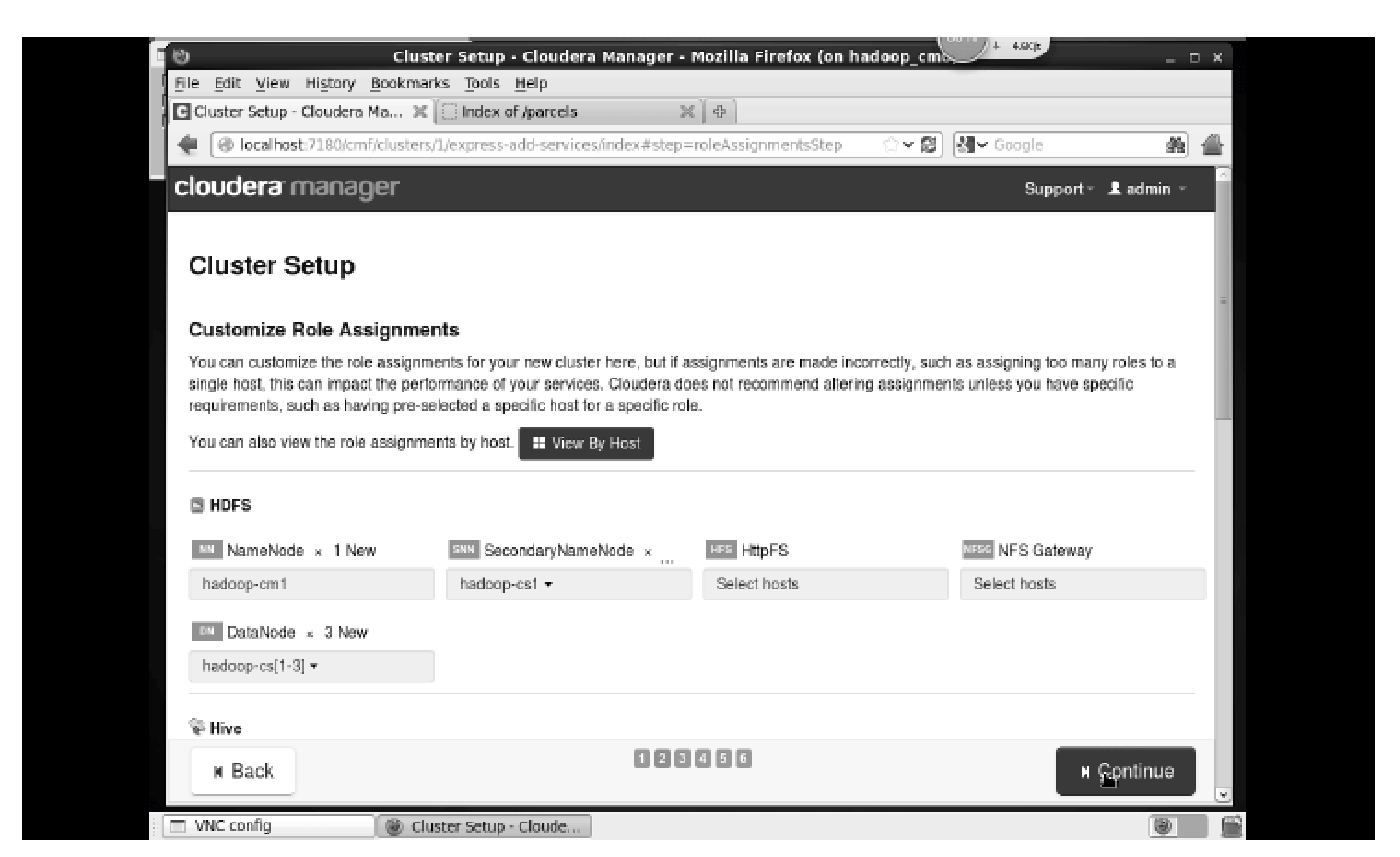
如果我们不需要像 Sqoop 这样的服务，我们也可以选择自定义安装方式 Custom Services。单击 Continue 按钮，打开如下图所示的界面。



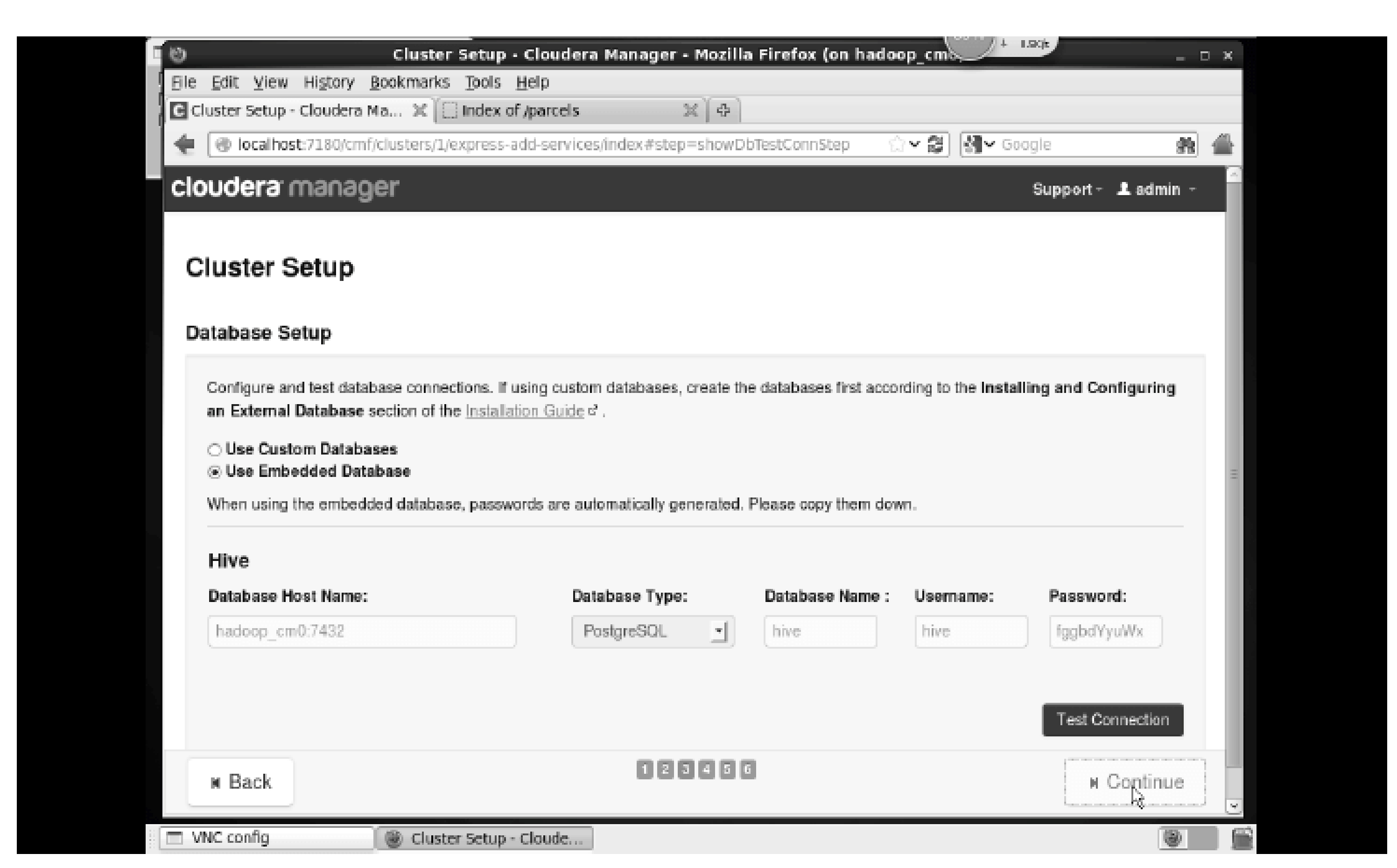
此处我们选择了需要的最小安装，也就是 HDFS，Hue，YARN，ZooKeeper。单击 Continue 按钮，界面如下图所示。



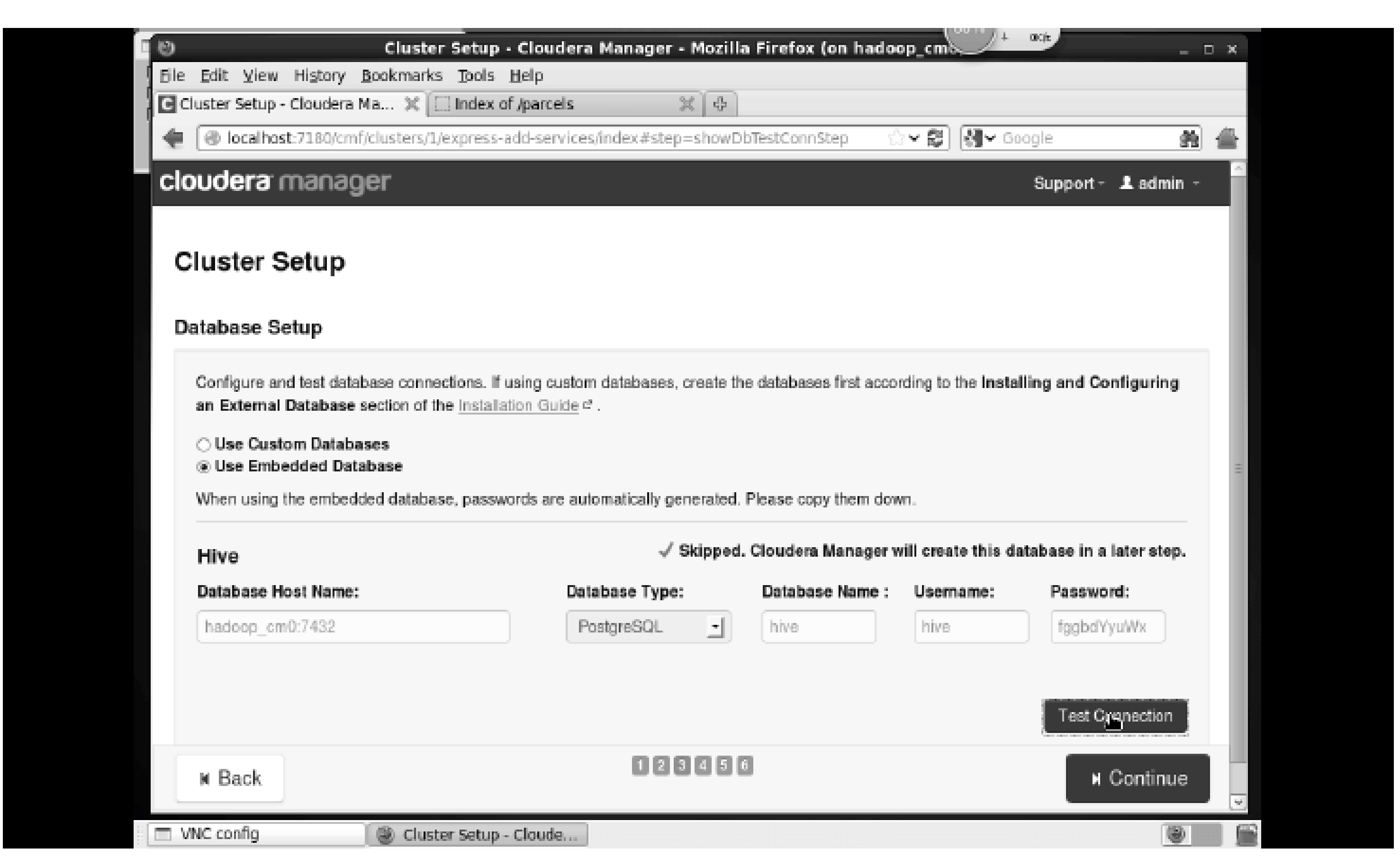
本步骤让我们为各节点选择角色，界面如下图所示。



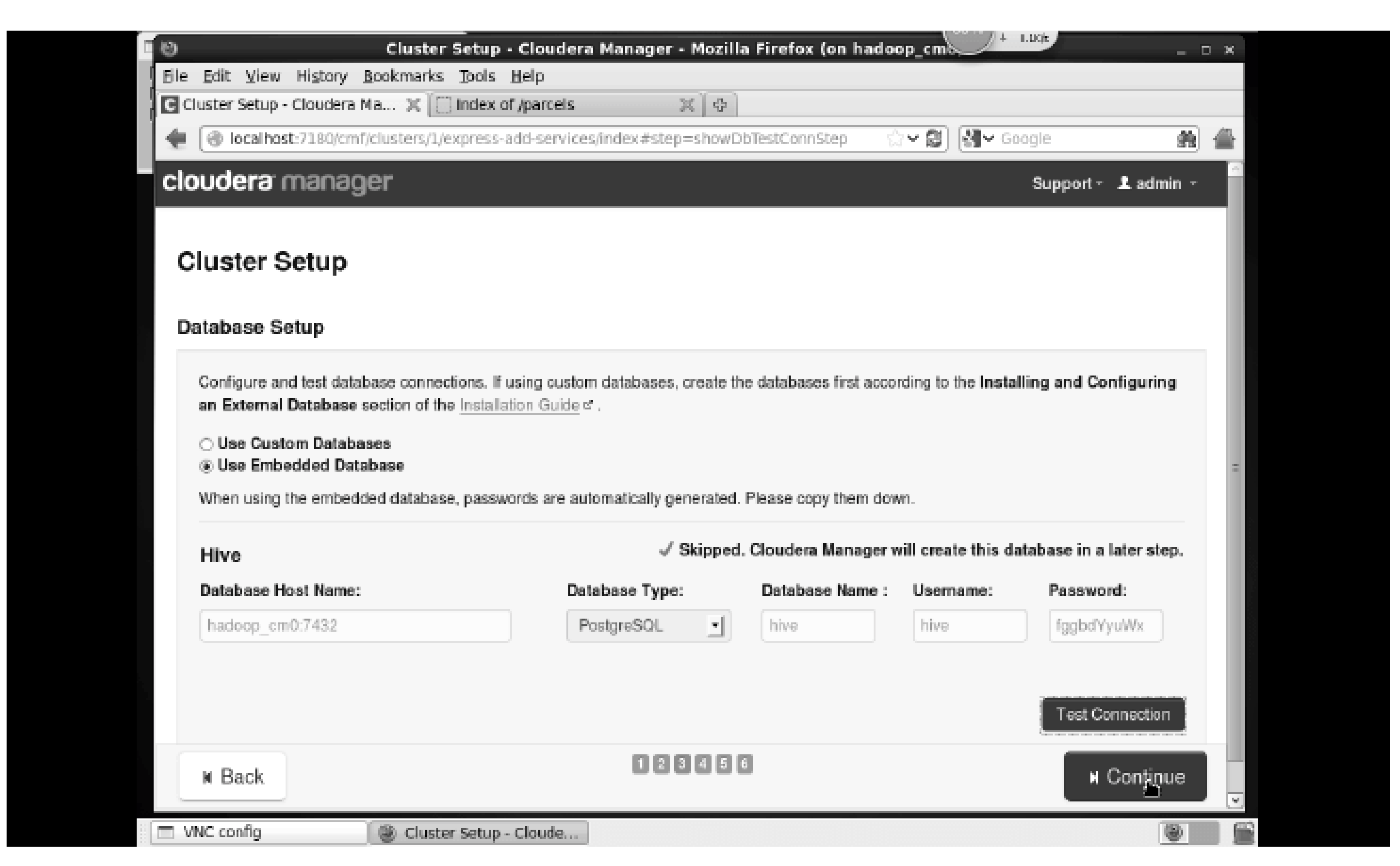
我们根据之前列中的规划，选择 hadoop-cm1 为 NameNode，hadoop-cs1 为 SecondaryNameNode，hadoop-cs1、hadoop-cs2、hadoop-cs3 为 DataNode。单击 Continue 按钮，界面如下图所示。



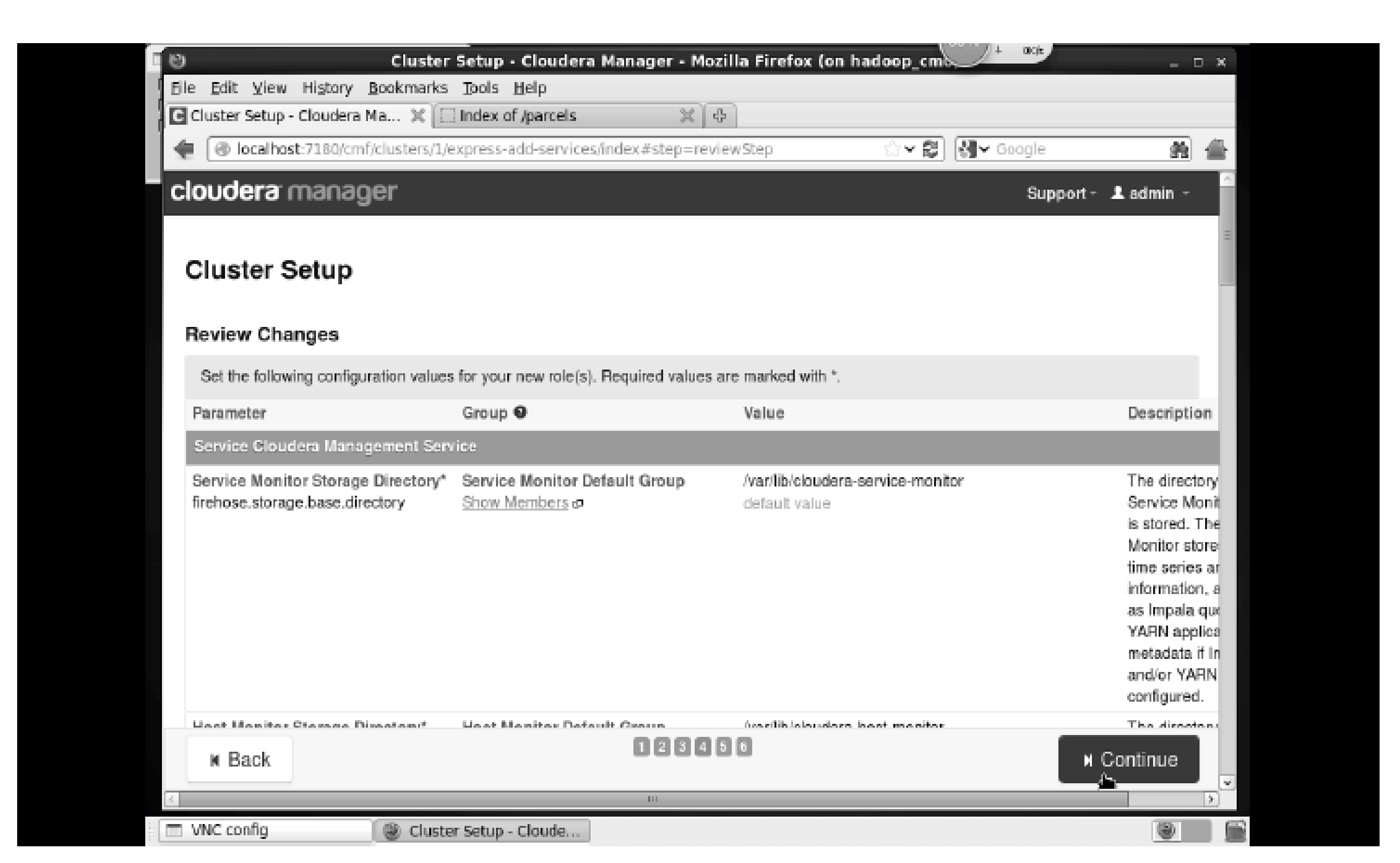
此处选择存储元数据使用的数据库，我们选择内置的 Postgresql 数据库，界面如下图所示。



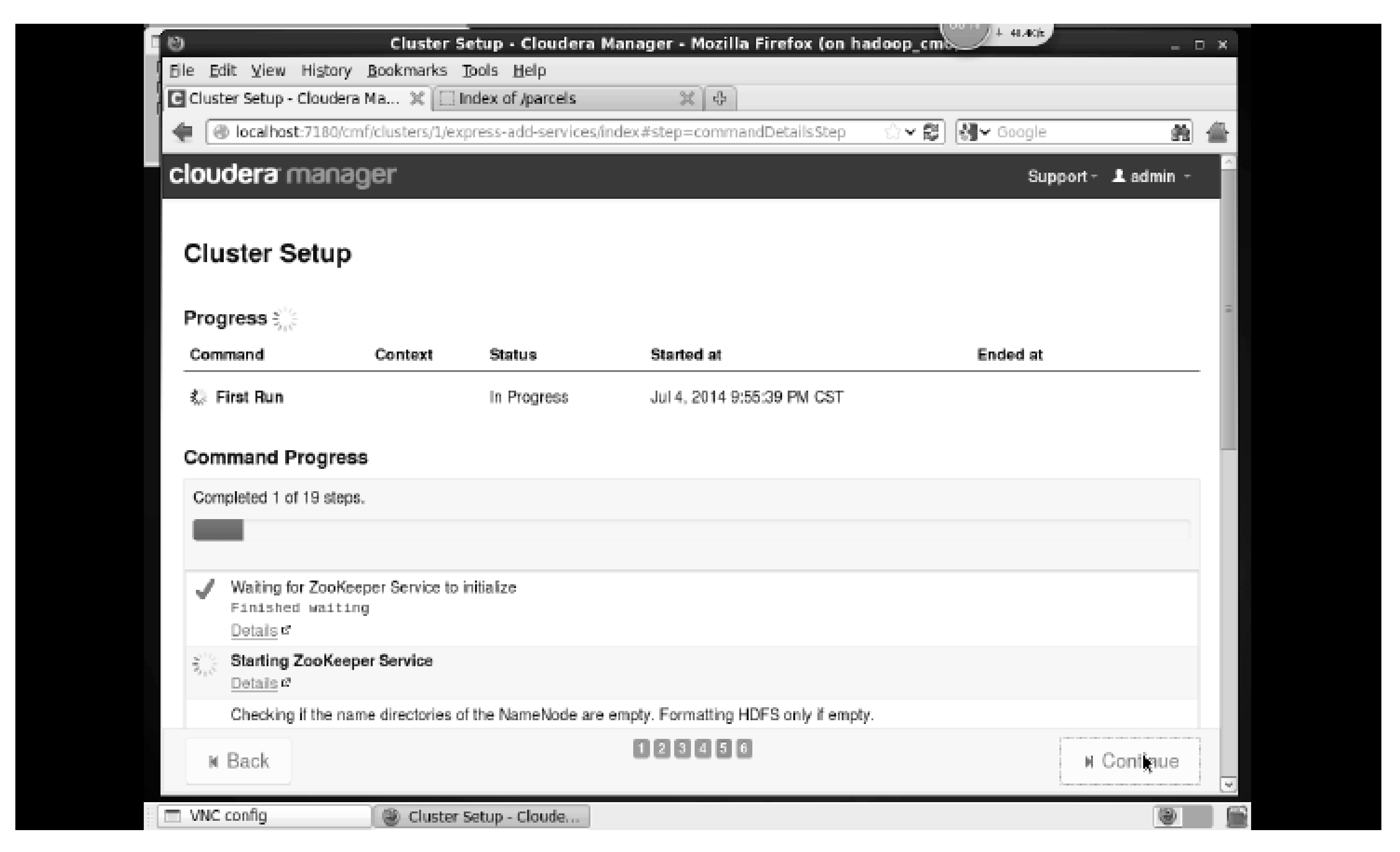
如果选择自定义的数据库，可以使用 Test Connection 按钮来确认数据库的连通性。但是如果选择内置的数据库，由于数据库还没有初始化，所以会忽略此步骤。界面如下图所示。



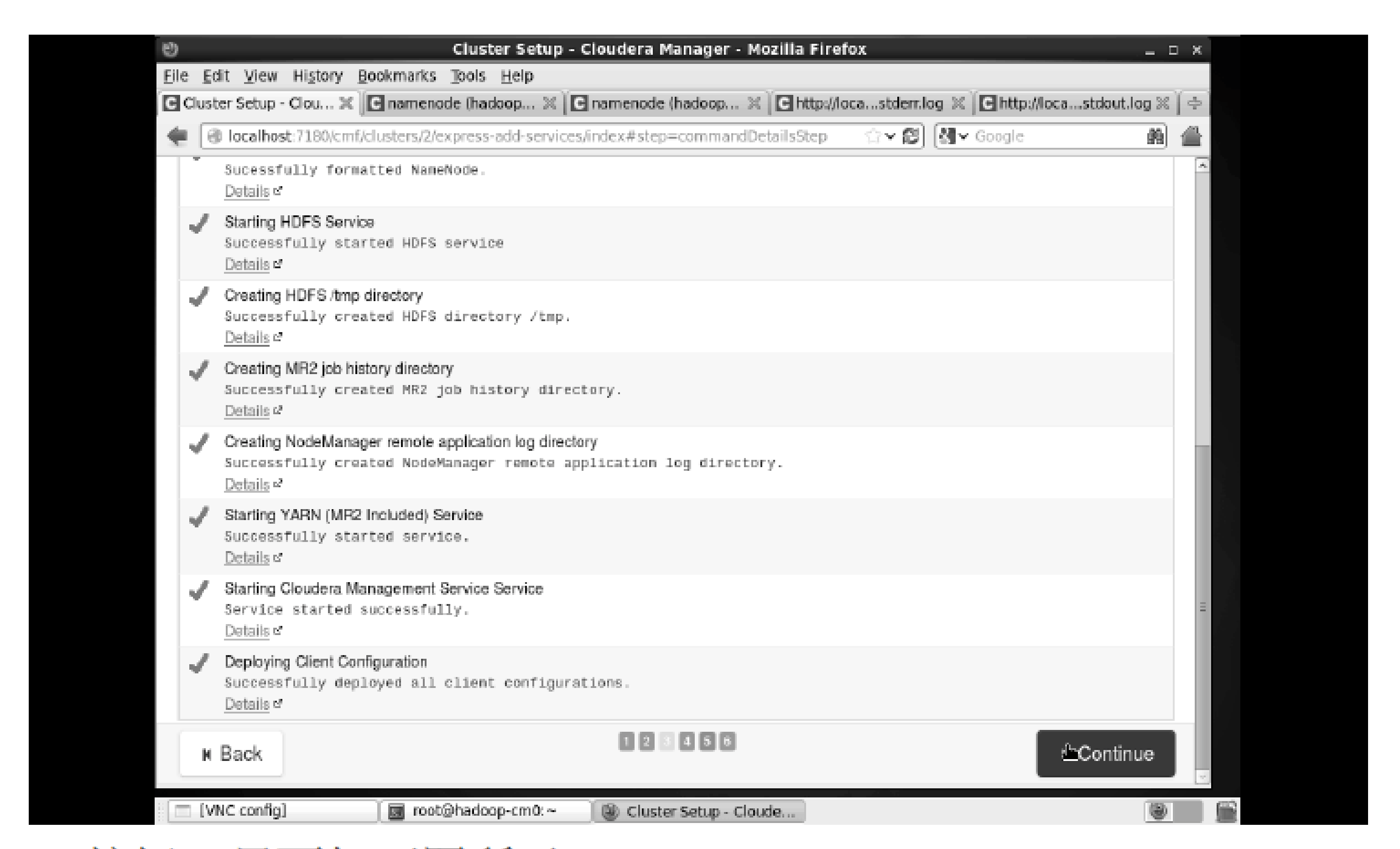
单击 Continue 按钮，界面如下图所示。



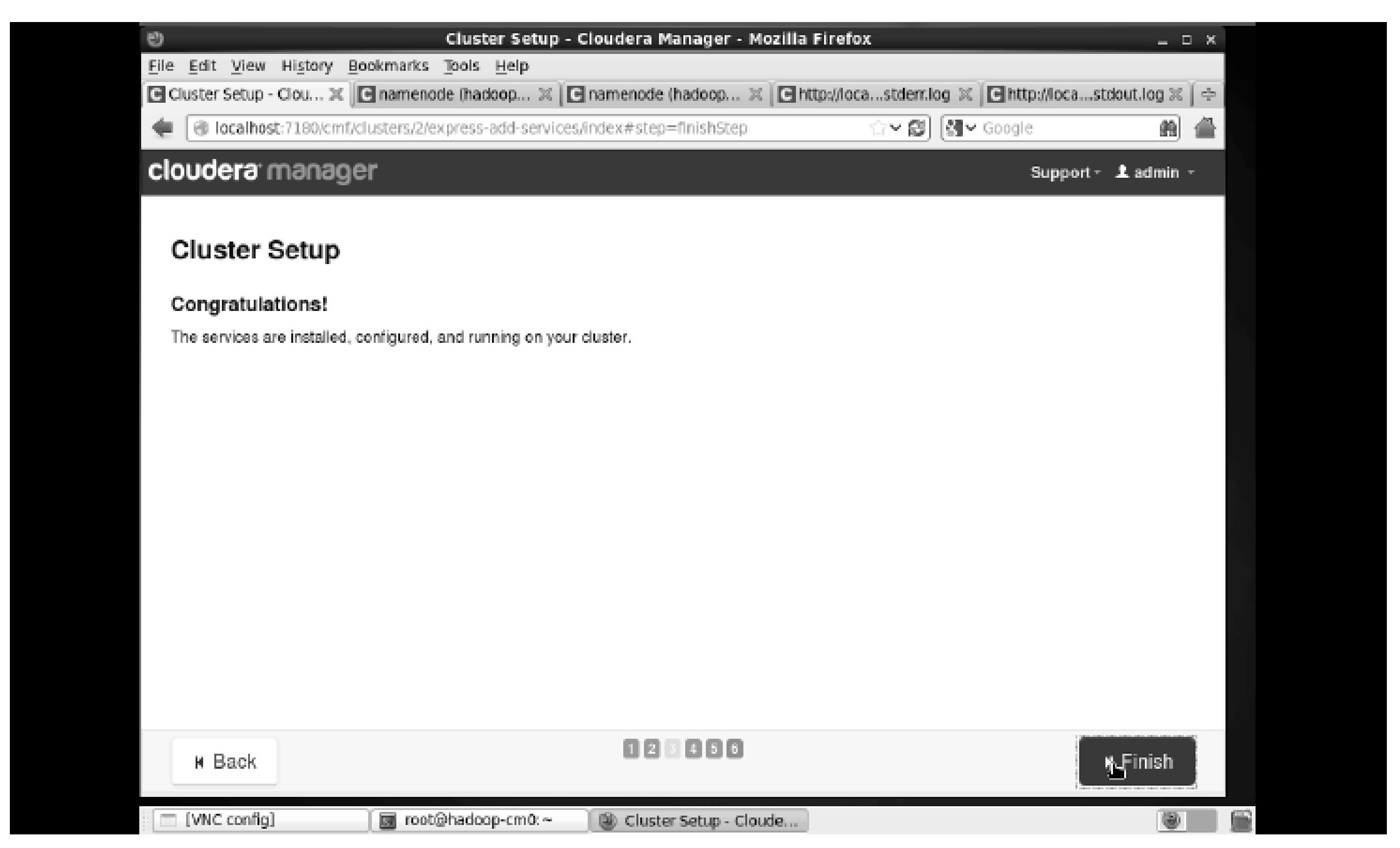
此处为集群的配置信息，我们可以指定 HDFS 在本地文件系统上对应的存储位置信息等。单击 Continue 按钮，CM 开始配置各服务，界面如下图所示。



配置完成后启动各服务，界面如下图所示。



单击 Continue 按钮，界面如下图所示。

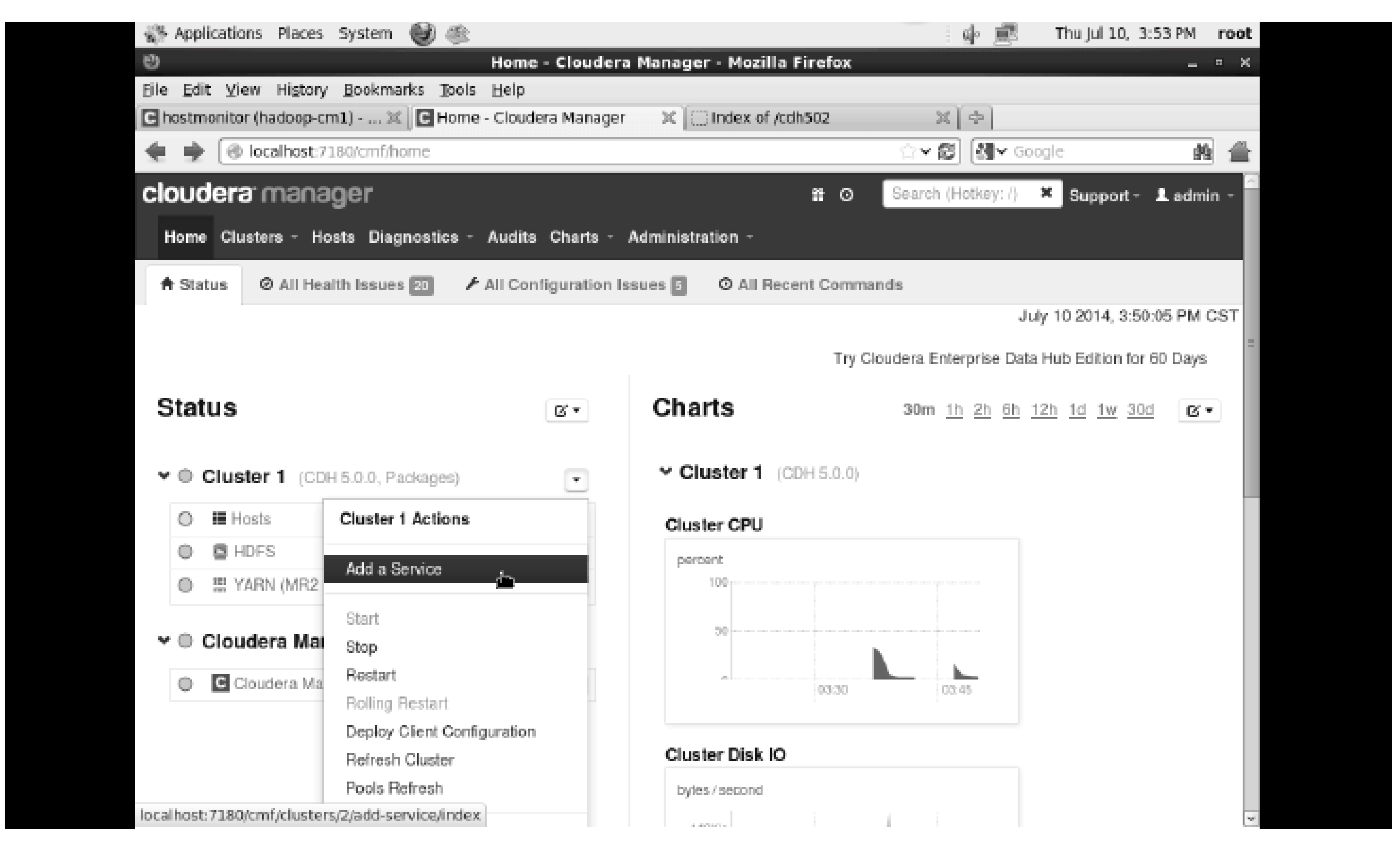


单击 Finish 按钮。CM 安装过程完成。

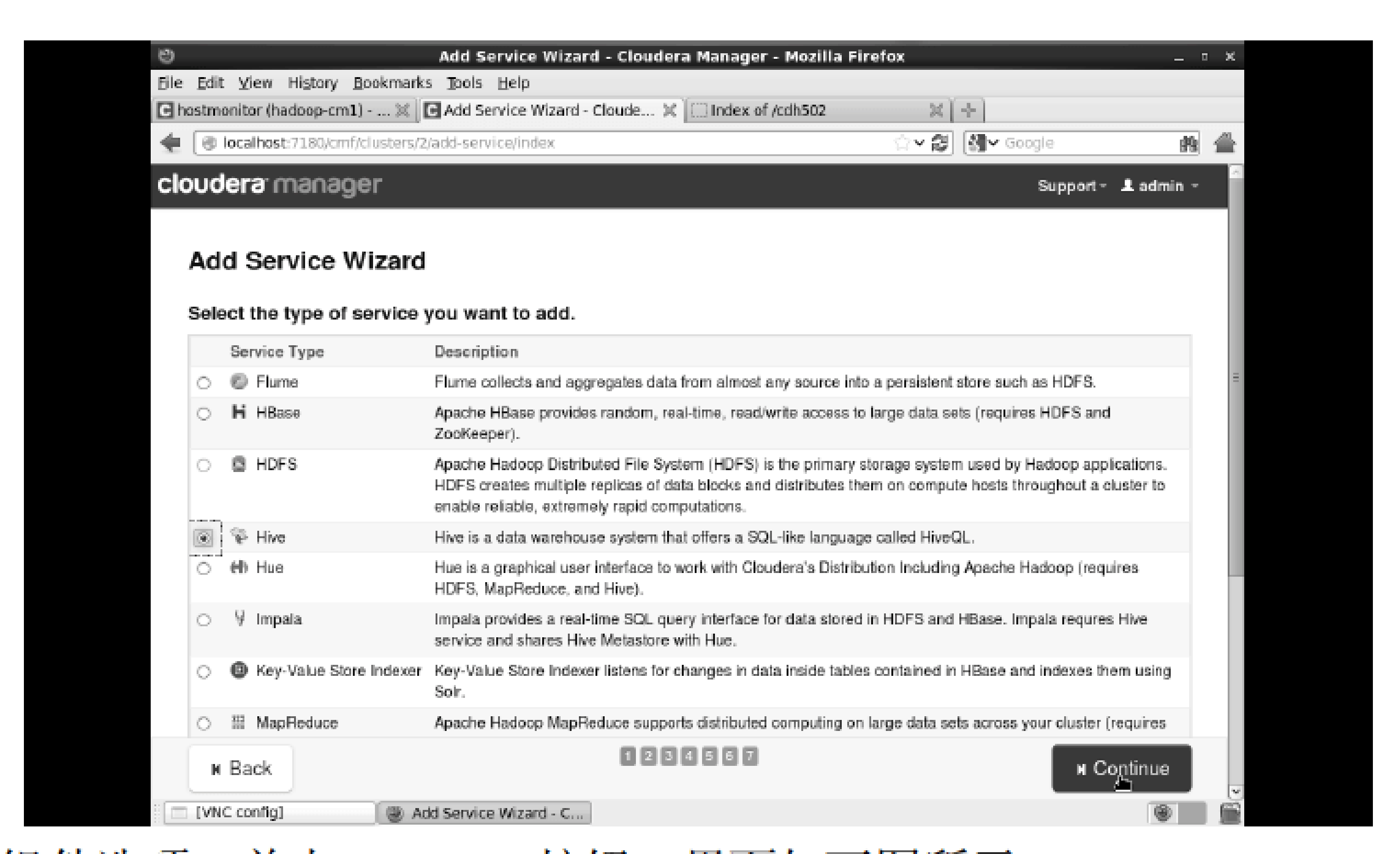
1.4 Hive 安装

Hive 是安装 Impala 之前的必选组件。除了 Impala 要部分共享 Hive 的元数据库之外，对于某些 Impala 支持得不是很好的数据文件格式，我们也需要先在 Hive 中加载数据，然后从 Impala 中进行查询。

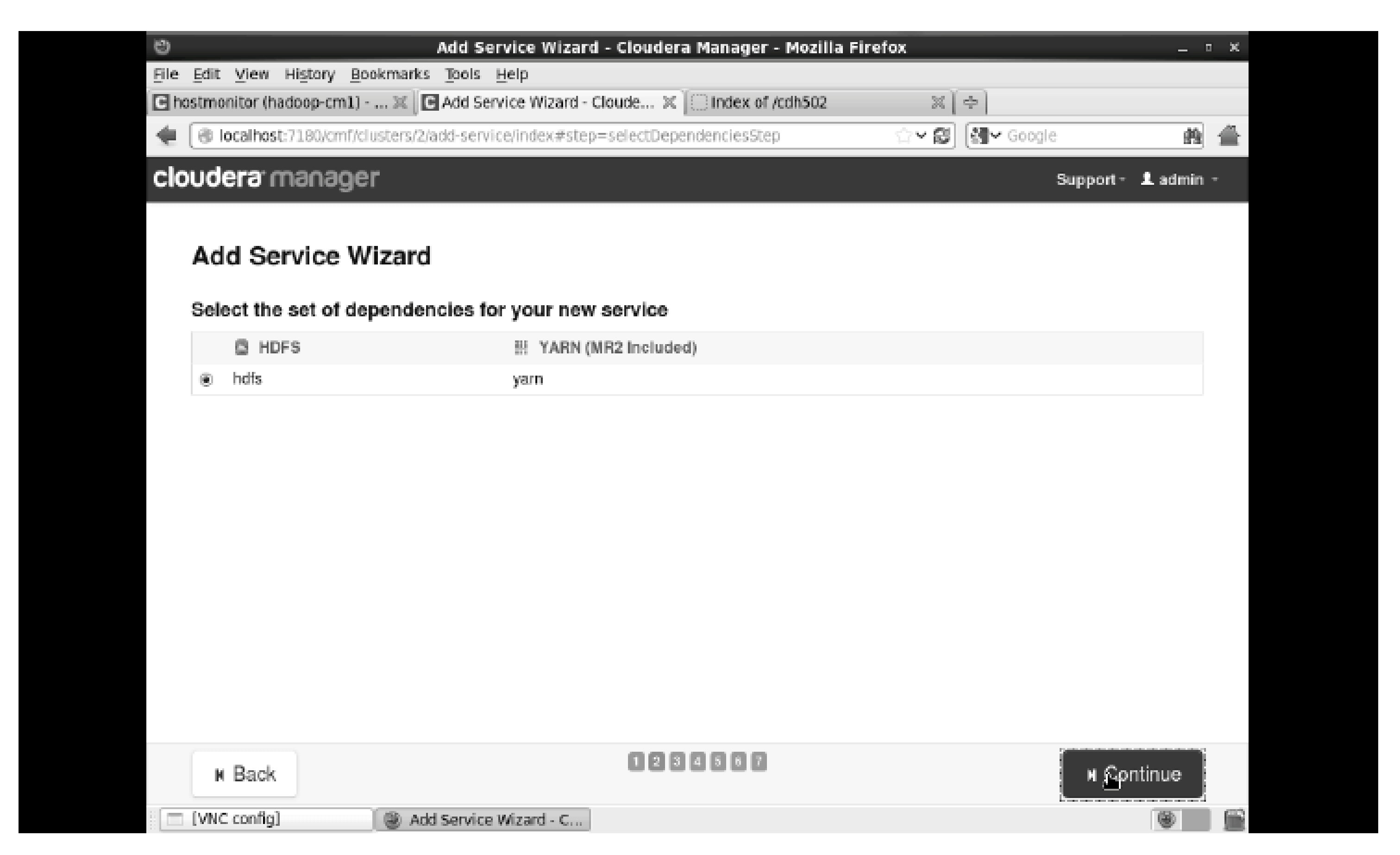
首先打开 CM 管理界面，如下图所示。



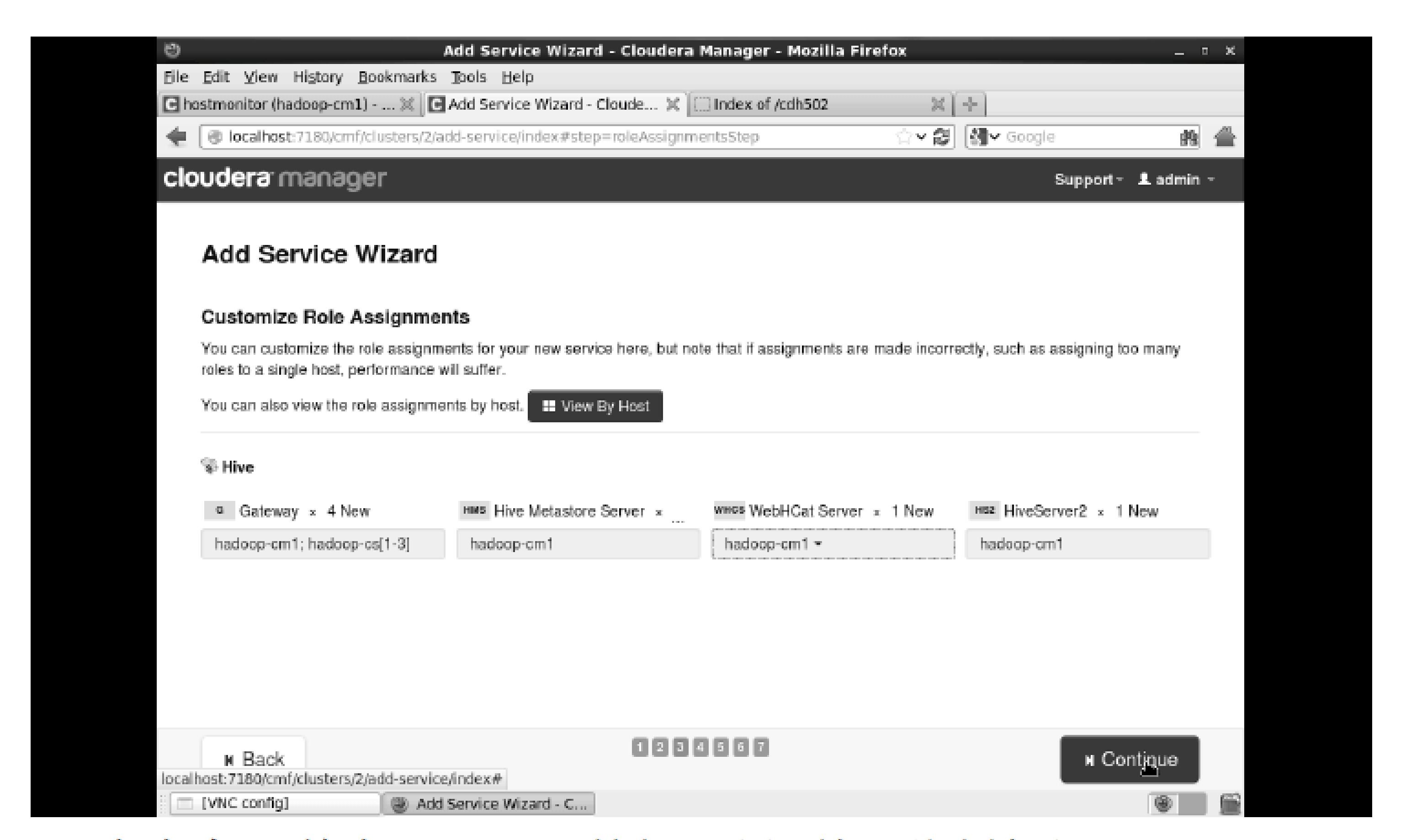
选择 Add a Service 菜单项，界面如下图所示。



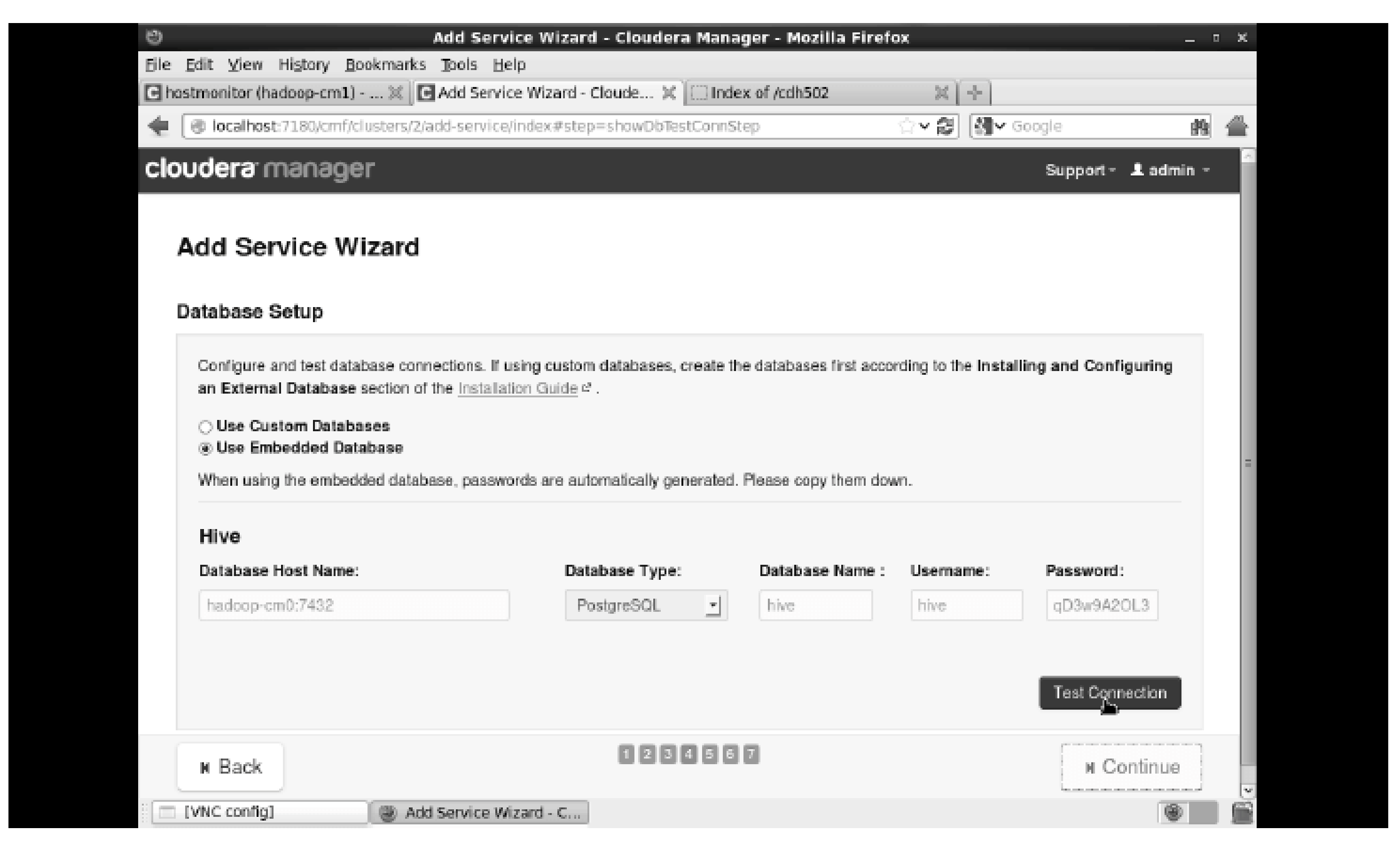
选择 Hive 组件选项。单击 Continue 按钮，界面如下图所示。



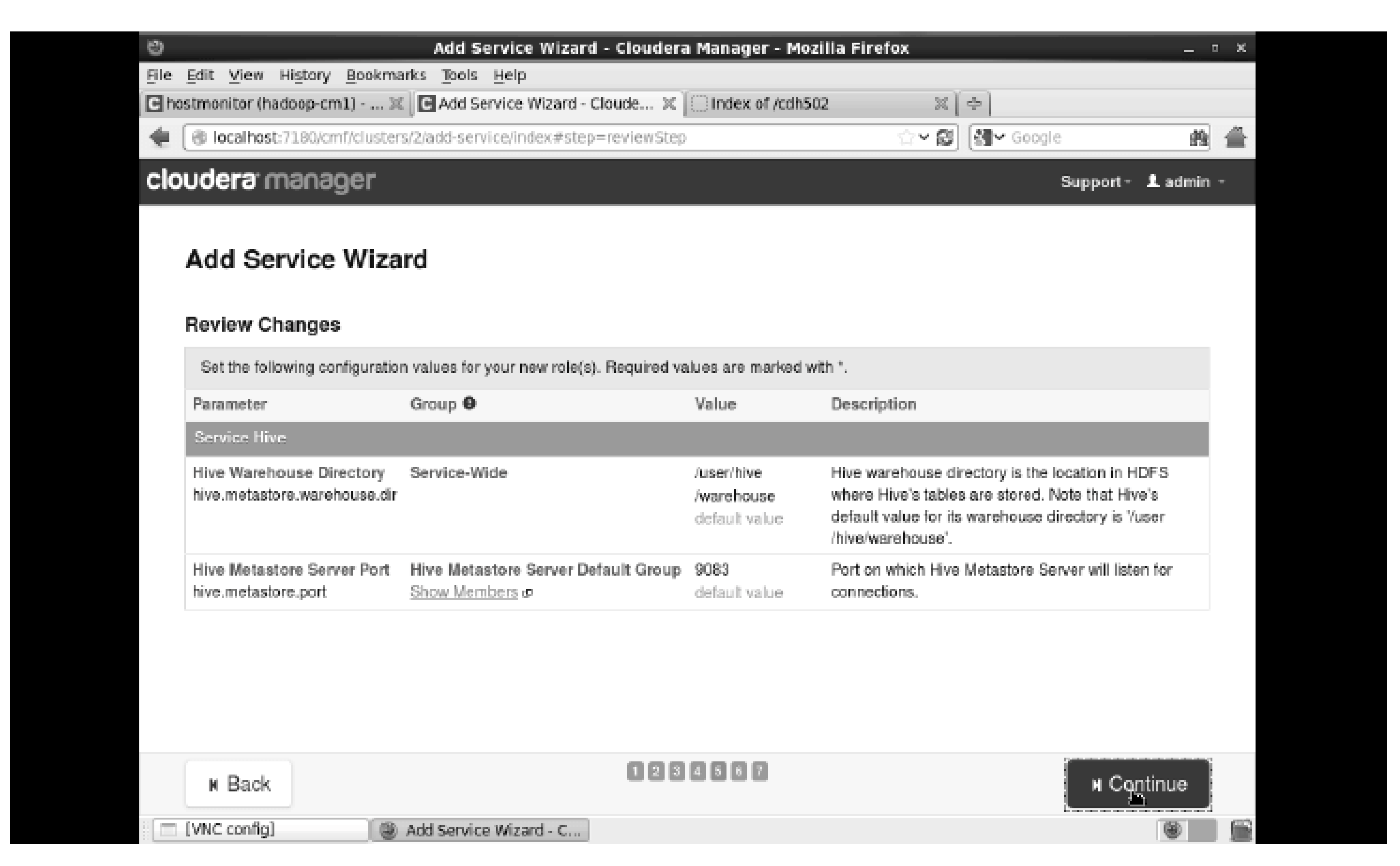
选择 Hive 依赖的服务，单击 Continue 按钮，界面如下图所示。



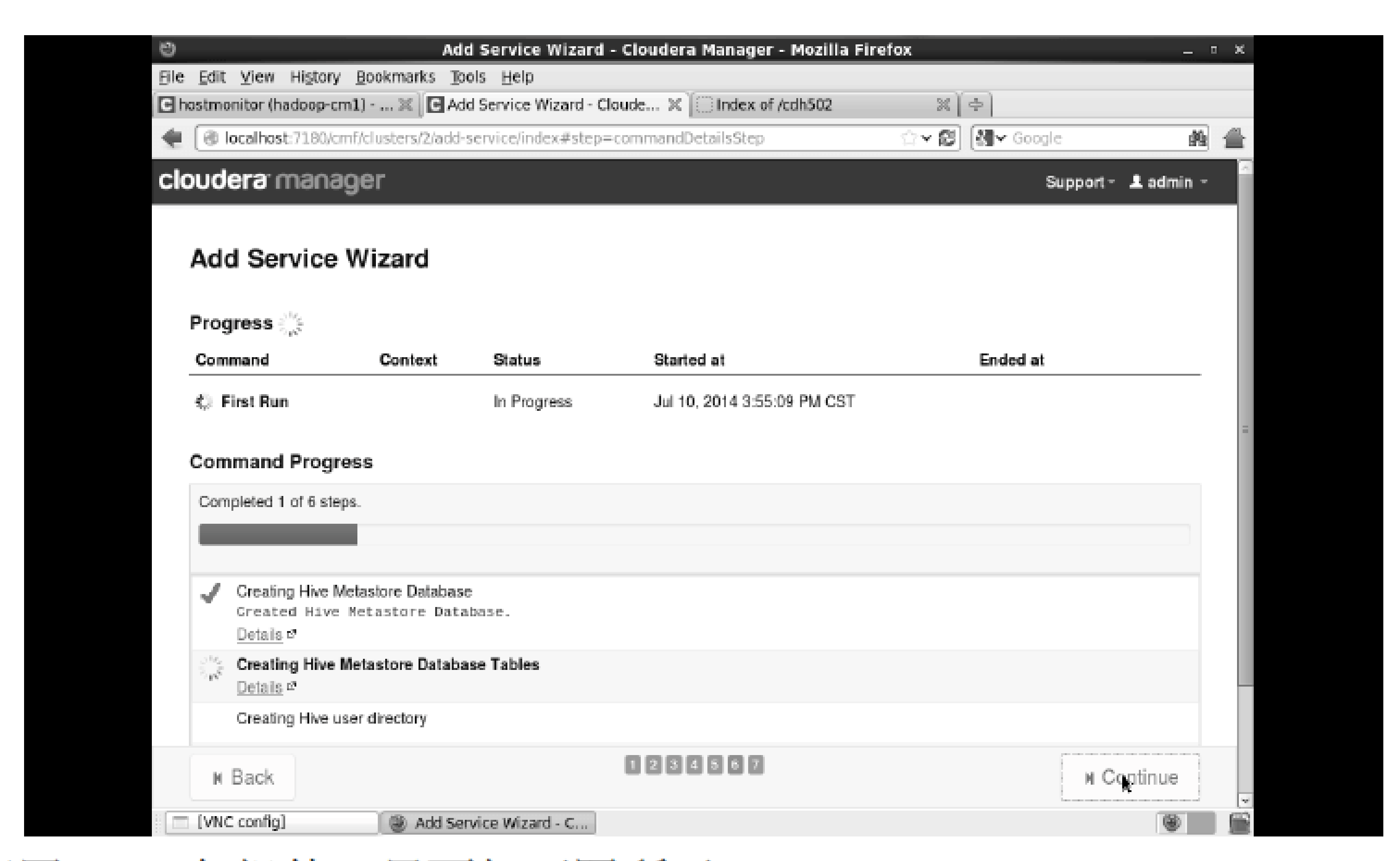
此处配置 Hive 各角色，单击 Continue 按钮，界面如下图所示。



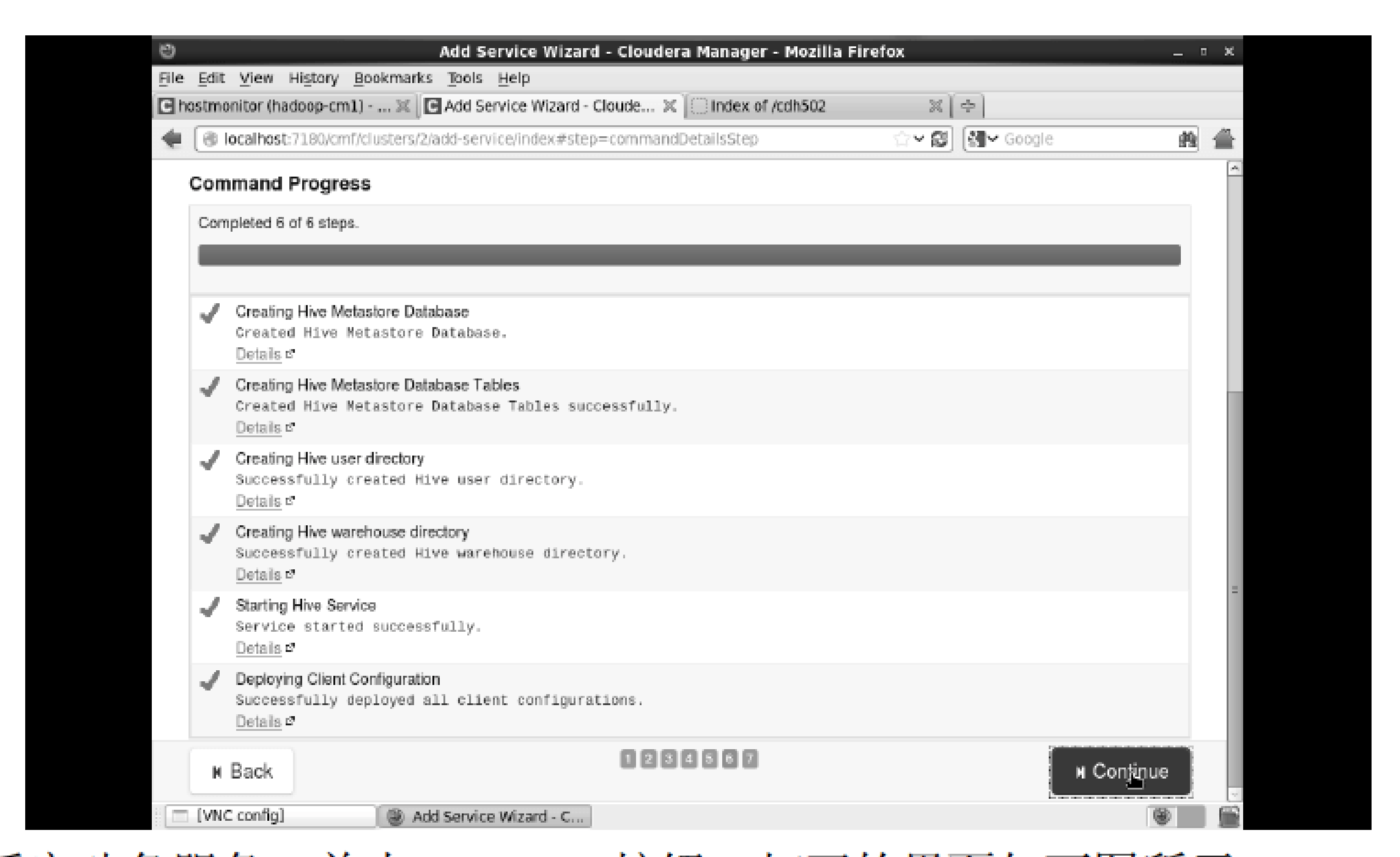
选择使用内嵌的数据库作为 Hive 的元数据库，打开的界面如下图所示。



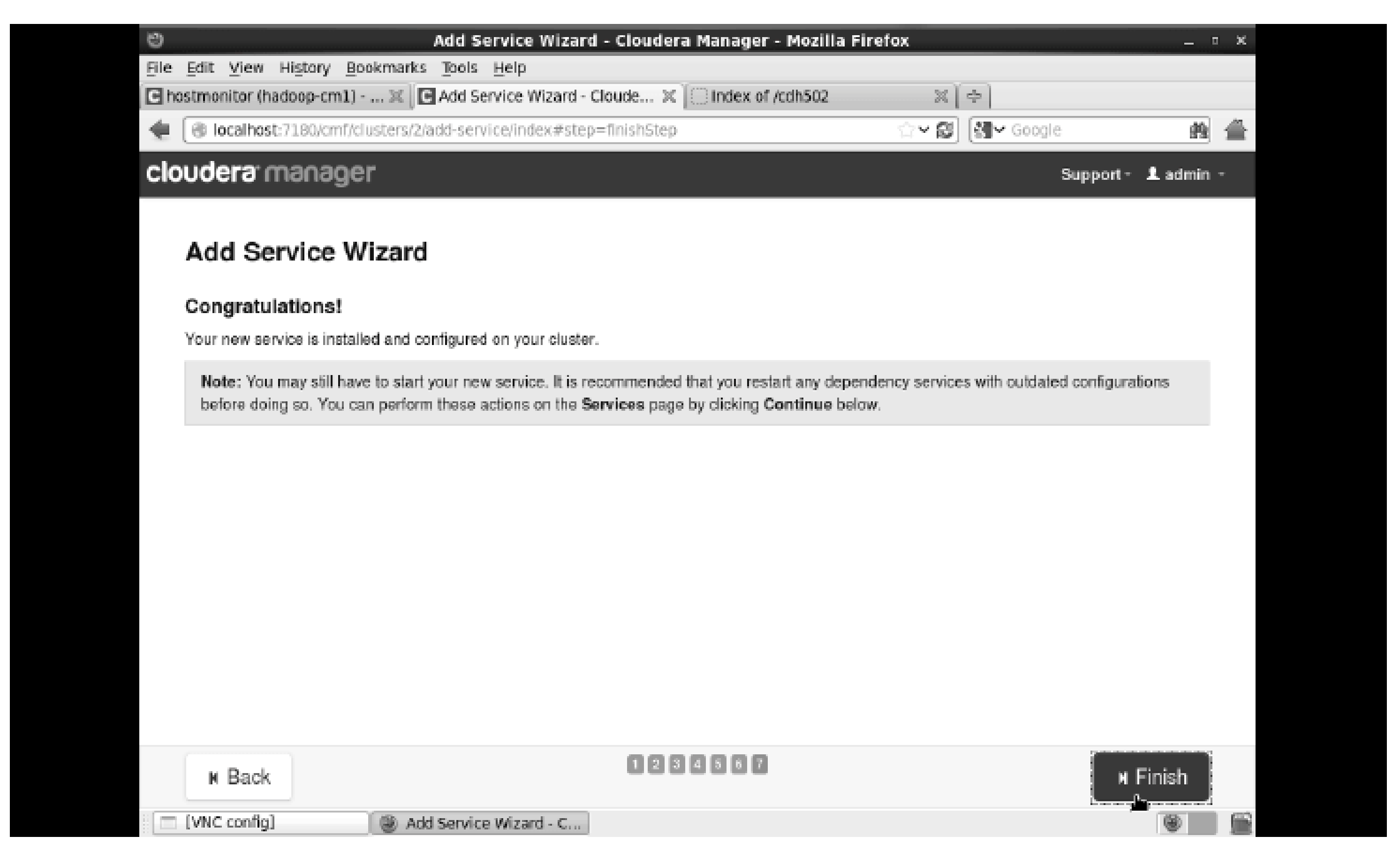
确认 Hive 的相关配置信息，单击 Continue 按钮，界面如下图所示。



CM 开始配置 Hive 各组件，界面如下图所示。



配置完成后启动各服务，单击 Continue 按钮，打开的界面如下图所示。



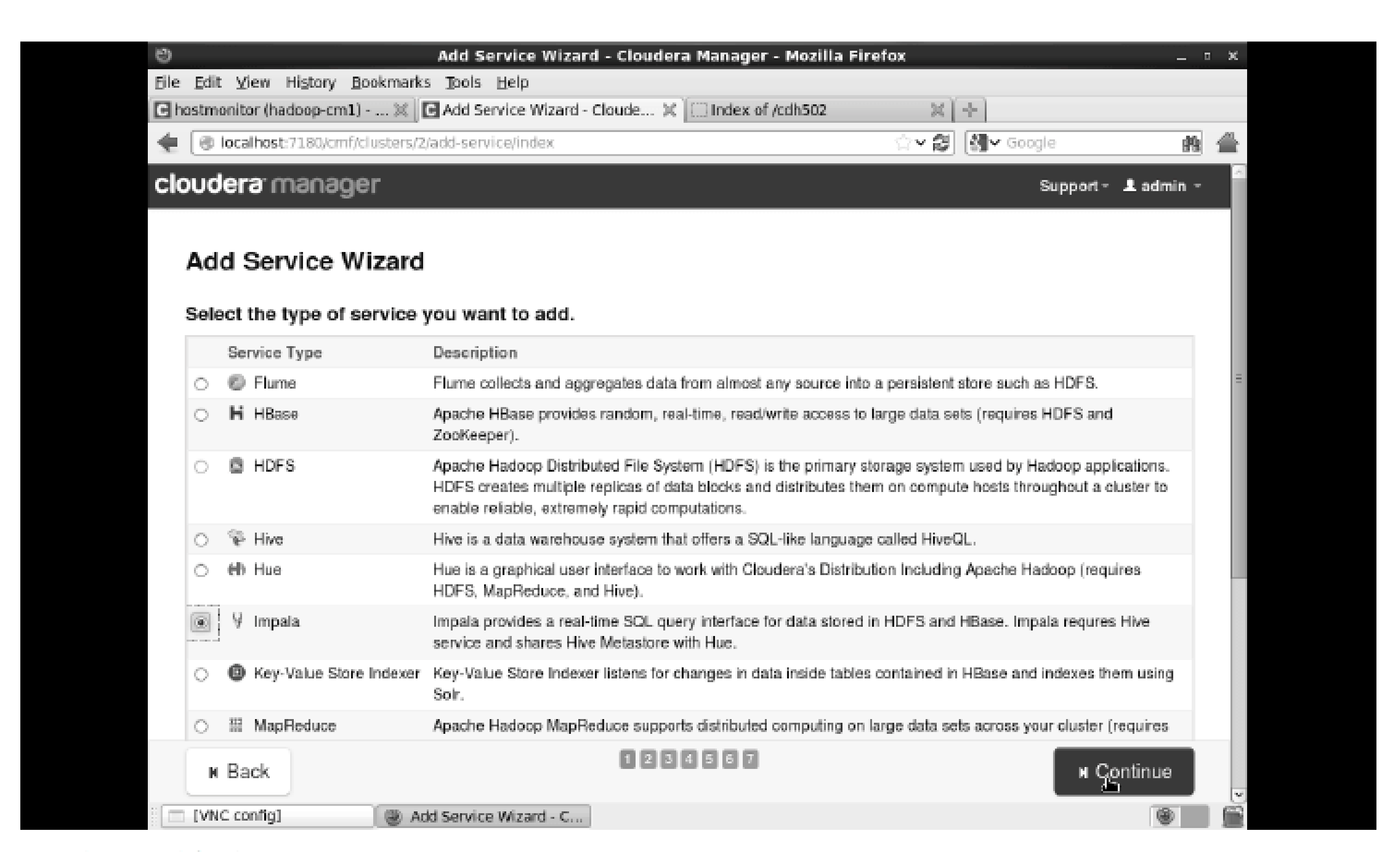
单击 Finish 按钮，Hive 安装完成。

1.5 Impala 安装

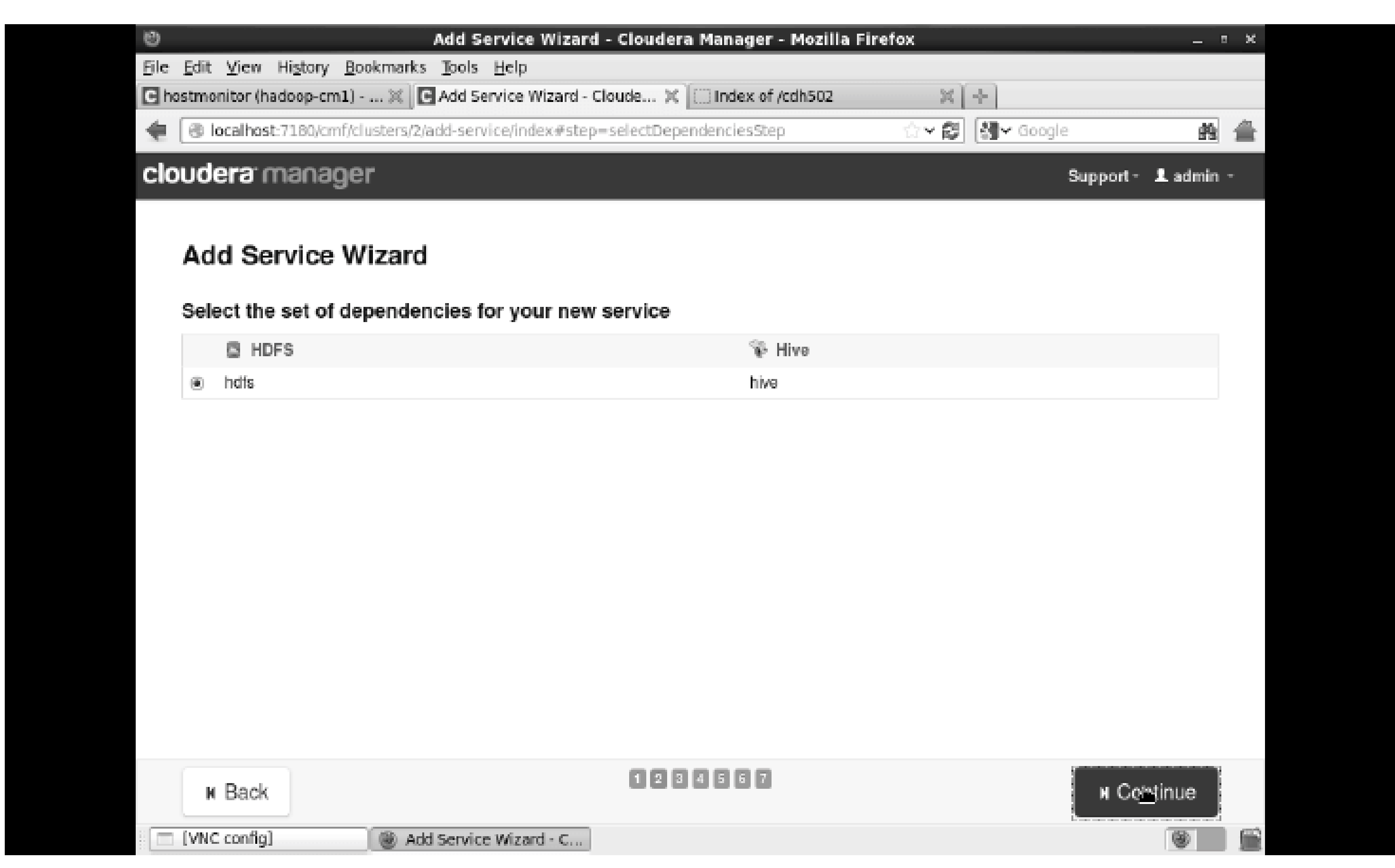
前面所有的准备工作都是为了最终安装 Impala 组件。Impala 的安装过程和 Hive 的安装过程非常类似，本节简单介绍如下。

在 CM 管理界面单击 Add a Service。

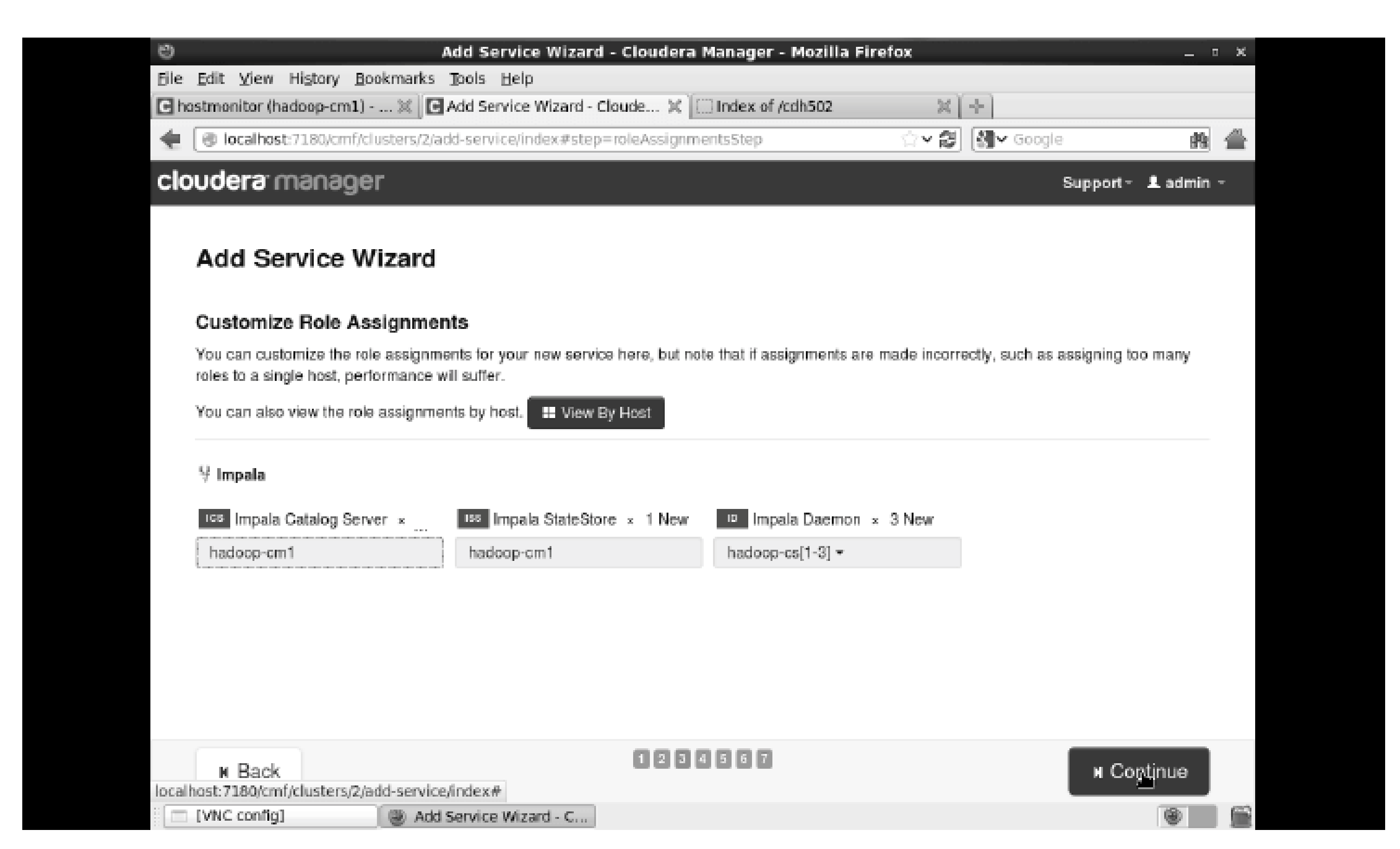
选择 Impala，单击 Continue 按钮。



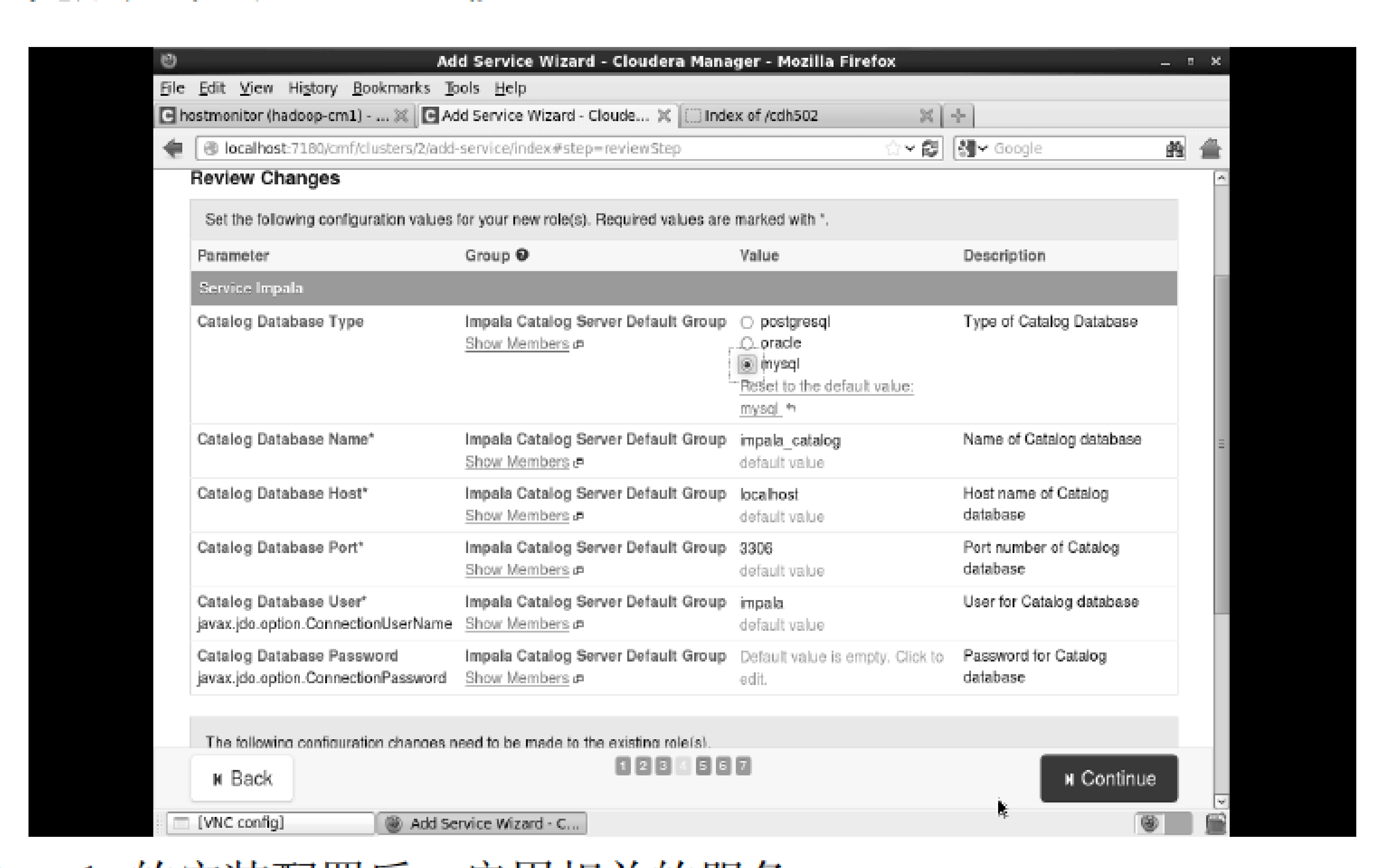
选择依赖的服务，单击 Continue 按钮。



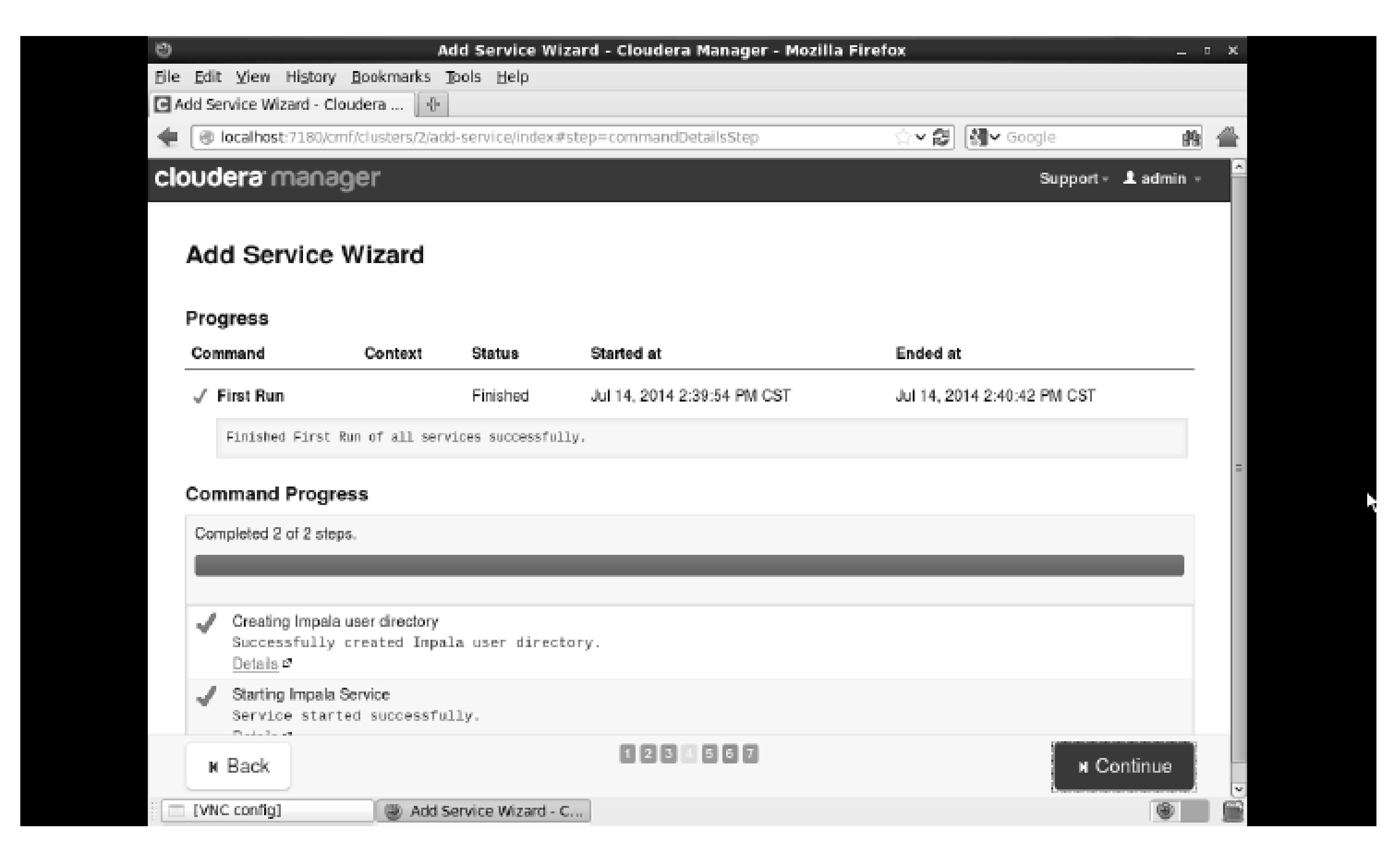
此处选择 Impala 角色，我们将 Catalog 服务和 StateStore 部署在主节点 hadoop-cm1 上，将 Impala 进程节点部署在从节点上。单击 Continue 按钮。



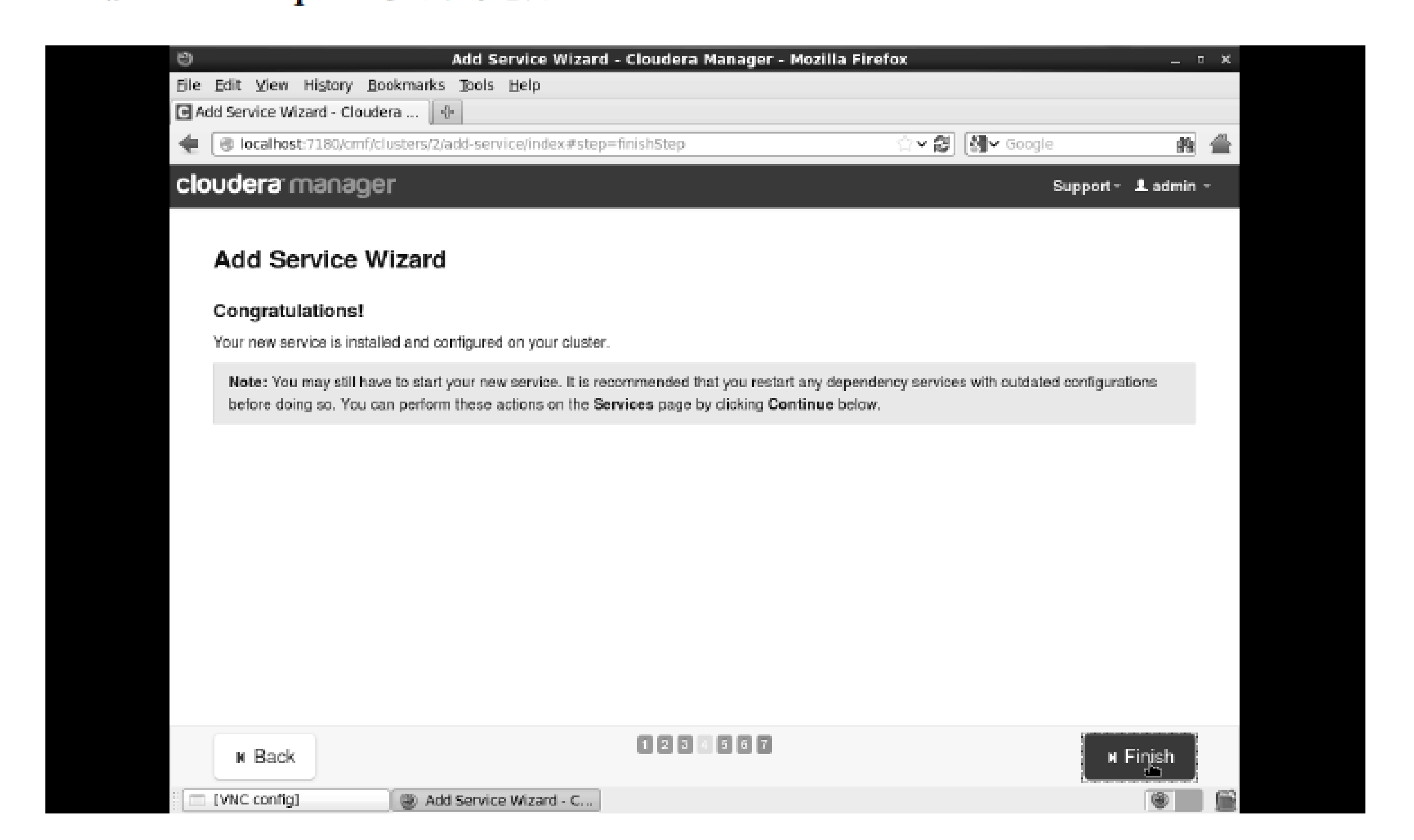
此处我们填写 Impala 元数据库信息。我们可以使用默认的 Postgresql 数据库，也可以使用其他数据库。配置完成后单击 Continue 按钮。



CM 完成 Impala 的安装配置后，启用相关的服务。



单击 Continue 按钮，Impala 安装完成。



第 2 章

Impala入门示例

在 Impala 安装完成之后，我们就可以来学习如何使用它了。本章我们将通过具体的实例直观地展示 Impala 的使用过程：数据加载，数据查询，分区表，外部分区表，笛卡尔连接及更新元数据等。

2.1 数据加载

如果我们想使用 Impala，那么首先需要将数据加载到 Impala 之中。如何才能将数据加载到 Impala 中呢？要想将数据加载到 Impala 当中，一共有两种途径：

- 使用外部表。此方式适用于我们已经有了数据文件的情况，我们只需将数据文件拷贝到 HDFS 上，然后建立一张 Impala 外部表，将外部表的存储位置指向数据文件的位置即可。
- 通过 INSERT 插入数据。此方式适用于我们没有数据文件，需要通过对其他表的数据进行过滤转换生成新的数据的情况。

如下将演示通过两种方式来向 Impala 中加载数据的过程。

1. 准备数据

本步骤需要将放在本地文件系统上的数据文件上传到 HDFS 的指定位置上。

放在本地文件系统上的原始数据文件如下所示：

tab1.csv

```
1,true,123.123,2012-10-24 08:55:00
2,false,1243.5,2012-10-25 13:40:00
3,false,24453.325,2008-08-22 09:33:21.123
4,false,243423.325,2007-05-12 22:32:21.33454
5,true,243.325,1953-04-22 09:11:33
```

tab2.csv

```
1,true,12789.123
2,false,1243.5
3,false,24453.325
4,false,2423.3254
5,true,243.325
60,false,243565423.325
70,true,243.325
80,false,243423.325
90,true,243.325
```

查看原始 HDFS 上 impala 的默认目录：

```
[root@hadoop-cm1 ~]# su - hdfs
-bash-4.1$ hdfs dfs -ls /user
Found 4 items
drwxr-xr-x   - hdfs   supergroup          0 2014-07-09 20:19 /user/hdfs
drwxrwxrwx   - mapred hadoop              0 2014-07-09 20:02 /user/history
drwxrwxr-t   - hive   hive               0 2014-07-10 15:55 /user/hive
drwxrwxr-x   - impala impala              0 2014-07-14 14:40 /user/impala
-bash-4.1$ hdfs dfs -ls /user/impala
```

为这两个文件分别建立单独的目录：

```
-bash-4.1$ hdfs dfs -mkdir -p /user/impala/tab1 /user/impala/tab2
-bash-4.1$ hdfs dfs -ls /user/impala
Found 2 items
drwxr-xr-x   - hdfs impala         0 2014-07-14 15:12 /user/impala/tab1
drwxr-xr-x   - hdfs impala         0 2014-07-14 15:12 /user/impala/tab2
```

将本地文件系统的数据文件上传到 HDFS 上：

```
-bash-4.1$ hdfs dfs -put tab1.csv /user/impala/tab1
-bash-4.1$ hdfs dfs -put tab2.csv /user/impala/tab2
-bash-4.1$ hdfs dfs -ls /user/impala/tab1
Found 1 items
-rw-r--r--   3 hdfs impala       193 2014-07-14 17:11 /user/impala/tab1/tab1.csv
-bash-4.1$ hdfs dfs -ls /user/impala/tab2
Found 1 items
-rw-r--r--   3 hdfs impala       158 2014-07-14 17:11 /user/impala/tab2/tab2.csv
-bash-4.1$
```

2. 创建表

此处我们创建三张表，分别为 tab1，tab2，tab3。其中 tab1，tab2 为外部表，直接使用 HDFS 上的数据文件。tab3 为内部表，通过表 tab1，tab2 的数据生成。

在 hadoop-cs1 节点上，连接到 impala-shell：

```
[root@hadoop-cs1 ~]# su - impala
-bash-4.1$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT
2014)
[hadoop-cs1:21000] >
```

在 impala-shell 中执行创建表的操作：

```
[hadoop-cs1:21000] > DROP TABLE IF EXISTS tab1;
Query: drop TABLE IF EXISTS tab1
[hadoop-cs1:21000] > CREATE EXTERNAL TABLE tab1
                   > (
                   > id INT,
                   > col_1 BOOLEAN,
                   > col_2 DOUBLE,
                   > col_3 TIMESTAMP
                   > )
                   > ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
                   > LOCATION '/user/impala/tab1';
Query: create EXTERNAL TABLE tab1 ( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3
TIMESTAMP ) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/user/impala/tab1'

Returned 0 row(s) in 3.81s
[hadoop-cs1:21000] > DROP TABLE IF EXISTS tab2;
Query: drop TABLE IF EXISTS tab2
[hadoop-cs1:21000] > CREATE EXTERNAL TABLE tab2
                   > (
                   > id INT,
```

```
                    > col_1 BOOLEAN,
                    > col_2 DOUBLE
                    > )
                    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
                    > LOCATION '/user/impala/tab2';
    Query: create EXTERNAL TABLE tab2 ( id INT, col_1 BOOLEAN, col_2 DOUBLE ) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' LOCATION '/user/impala/tab2'

    Returned 0 row(s) in 0.23s
```

tab1、tab2 两张表创建为外部表，并将数据位置指向了我们之前创建的 HDFS 目录。这里值得我们注意的是，我们指定的位置是一个目录，而不是文件，如果这个目录中包含其他文件，无论这个文件的名称是什么，Impala 会将其统统作为表的数据文件。在创建表时，使用了 EXTERNAL 关键字，它的含义与其他传统的关系型数据库的外部表意义类似，表的定义保存在 Impala 中，使用的数据文件就是定义在 LOCATION 中的 HDFS 上对应位置的数据文件。在 ROW FORMAT DELIMITED 子句中指定数据文件使用逗号分隔如下例所示。

```
    [hadoop-cs1:21000] > DROP TABLE IF EXISTS tab3;
    Query: drop TABLE IF EXISTS tab3
    [hadoop-cs1:21000] > CREATE TABLE tab3
                    > (
                    > id INT,
                    > col_1 BOOLEAN,
                    > col_2 DOUBLE,
                    > month INT,
                    > day INT
                    > )
                    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
    Query: create TABLE tab3 ( id INT, col_1 BOOLEAN, col_2 DOUBLE, month INT, day INT )
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    Returned 0 row(s) in 0.13s
```

tab3 创建为内部表。该创建表的过程没有使用 EXTERNAL 关键字。存储时使用 Impala 默认的数据文件位置。存储的数据文件也是以逗号分隔。

在执行完创建表的过程后，我们如何查看表定义信息呢？可以使用 DESCRIBE table_name 语句查看已经创建过的表的定义信息：

```
    [hadoop-cs1:21000] > show tables;
    Query: show tables
    +-------+
```

```
| name |
+-------+
| tab1 |
| tab2 |
| tab3 |
+-------+
Returned 3 row(s) in 0.01s
```

通过该命令，我们可以查看创建过的所有的表。

```
[hadoop-cs1:21000] > desc tab1;
Query: describe tab1
+--------+-----------+---------+
| name  | type      | comment |
+--------+-----------+---------+
| id    | int       |         |
| col_1 | boolean   |         |
| col_2 | double    |         |
| col_3 | timestamp |         |
+--------+-----------+---------+
Returned 4 row(s) in 0.11s
[hadoop-cs1:21000] > desc tab2;
Query: describe tab2
+--------+---------+---------+
| name  | type    | comment |
+--------+---------+---------+
| id    | int     |         |
| col_1 | boolean |         |
| col_2 | double  |         |
+--------+---------+---------+
Returned 3 row(s) in 0.01s
[hadoop-cs1:21000] > desc tab3;
Query: describe tab3
+--------+---------+---------+
| name  | type    | comment |
+--------+---------+---------+
| id    | int     |         |
| col_1 | boolean |         |
| col_2 | double  |         |
| month | int     |         |
```

```
| day    | int      |          |
+--------+---------+---------+
Returned 5 row(s) in 1.21s
```

DESCRIBE 命令可以简写为 DESC。通过该命令可以查看表的列名称及列定义等信息。

3. 加载数据

对于表 tab1，tab2，由于他们是外部表，所以我们只需要通过表定义与数据文件的位置关联起来即可，不需要额外的数据加载过程。

```
[hadoop-cs1:21000] > select * from tab1;
Query: select * from tab1
+----+-------+------------+-------------------------------+
| id | col_1 | col_2      | col_3                         |
+----+-------+------------+-------------------------------+
| 1  | true  | 123.123    | 2012-10-24 08:55:00           |
| 2  | false | 1243.5     | 2012-10-25 13:40:00           |
| 3  | false | 24453.325  | 2008-08-22 09:33:21.123000000 |
| 4  | false | 243423.325 | 2007-05-12 22:32:21.334540000 |
| 5  | true  | 243.325    | 1953-04-22 09:11:33           |
+----+-------+------------+-------------------------------+
Returned 5 row(s) in 1.14s
[hadoop-cs1:21000] > select * from tab2;
Query: select * from tab2
+----+-------+---------------+
| id | col_1 | col_2         |
+----+-------+---------------+
| 1  | true  | 12789.123     |
| 2  | false | 1243.5        |
| 3  | false | 24453.325     |
| 4  | false | 2423.3254     |
| 5  | true  | 243.325       |
| 60 | false | 243565423.325 |
| 70 | true  | 243.325       |
| 80 | false | 243423.325    |
| 90 | true  | 243.325       |
+----+-------+---------------+
Returned 9 row(s) in 0.64s
```

而此时，表 tab3 是没有数据的。

```
[hadoop-cs1:21000] > select * from tab3;
Query: select * from tab3

Returned 0 row(s) in 1.40s
```

我们可以通过 INSERT 语句将表 tab1，tab2 的数据经过关联，转换插入到表 tab3 中。

```
[hadoop-cs1:21000] > INSERT OVERWRITE TABLE tab3
                   > SELECT id, col_1, col_2, MONTH(col_3), DAYOFMONTH(col_3)
                   > FROM tab1 WHERE YEAR(col_3) = 2012;
Query: insert OVERWRITE TABLE tab3 SELECT id, col_1, col_2, MONTH(col_3),
DAYOFMONTH(col_3) FROM tab1 WHERE YEAR(col_3) = 2012
Inserted 2 rows in 1.12s
```

这里的 INSERT … SELECT 语句与传统数据库中的基本相同。OVERWRITE 关键字表示将使用查询的结果覆盖表中已经存在的数据。

```
[hadoop-cs1:21000] > select * from tab3;
Query: select * from tab3
+----+-------+---------+-------+-----+
| id | col_1 | col_2   | month | day |
+----+-------+---------+-------+-----+
| 1  | true  | 123.123 | 10    | 24  |
| 2  | false | 1243.5  | 10    | 25  |
+----+-------+---------+-------+-----+
Returned 2 row(s) in 0.44s
```

插入之后，我们即可从表 tab3 中查询出数据。需要说明的是，Impala 不具备传统数据库的 ACID 属性，插入数据后不需要做 COMMIT 操作。

由于 Impala 表的数据都是存储在 HDFS 上的，所以进行完插入操作之后，Impala 会自动为内部表 tab3 创建一个数据目录，并将数据文件写入其中。

```
[root@hadoop-cm1 ~]#  hdfs dfs -ls /user/hive/warehouse/tab3
Found 2 items
drwxrwxrwt   -   impala   hive          0   2014-11-18   12:46
/user/hive/warehouse/tab3/.impala_insert_staging
-rw-r--r--   3   impala   hive         42   2014-11-18   12:46
/user/hive/warehouse/tab3/1f49071d98a6ccc2-fdb73a70ab90808d_264960562_data.0
```

2.2 数据查询

Impala 支持大部分传统数据库支持的聚集，关联，子查询等操作。

1. 聚集和关联操作

```
[hadoop-cs1:21000] > SELECT tab1.col_1, MAX(tab2.col_2), MIN(tab2.col_2)
                  > FROM tab2 JOIN tab1 USING (id)
                  > GROUP BY col_1 ORDER BY 1 LIMIT 5;
Query: select tab1.col_1, MAX(tab2.col_2), MIN(tab2.col_2) FROM tab2 JOIN tab1 USING (id) GROUP BY col_1 ORDER BY 1 LIMIT 5
+-------+-----------------+-----------------+
| col_1 | max(tab2.col_2) | min(tab2.col_2) |
+-------+-----------------+-----------------+
| false | 24453.325       | 1243.5          |
| true  | 12789.123       | 243.325         |
+-------+-----------------+-----------------+
Returned 2 row(s) in 0.83s
```

对表 tab1 和 tab2 使用 id 进行关联，并按列 tab1.col_1 进行分组操作，并计算出对应的列 tab2.col_2 对应的最大值和最小值。

2. 带有子查询的聚集和关联操作

```
[hadoop-cs1:21000] > SELECT tab2.*
                  > FROM tab2,
                  > (SELECT tab1.col_1, MAX(tab2.col_2) AS max_col2
                  > FROM tab2, tab1
                  > WHERE tab1.id = tab2.id
                  > GROUP BY col_1) subquery1
                  > WHERE subquery1.max_col2 = tab2.col_2;
Query: select tab2.* FROM tab2, (SELECT tab1.col_1, MAX(tab2.col_2) AS max_col2 FROM tab2, tab1 WHERE tab1.id = tab2.id GROUP BY col_1) subquery1 WHERE subquery1.max_col2 = tab2.col_2
+----+-------+-----------+
| id | col_1 | col_2     |
+----+-------+-----------+
| 1  | true  | 12789.123 |
| 3  | false | 24453.325 |
+----+-------+-----------+
```

```
Returned 2 row(s) in 0.92s
```

对表 tab1，tab2 按照 id 进行关联，tab1.col_1 进行分组，计算 tab2.col_2 的最大值，并将语句作为子查询与 tab2 关联。

2.3 分区表

与传统的数据库类似，Impala 的分区表也会将某个分区的数据单独存放，当我们指定的 WHERE 条件是针对某个分区的查询时，Impala 将只会扫描该分区的数据文件，大大减少了需要从磁盘读取的数据量，提高了查询的效率。

本节内容我们将介绍如何创建一张分区表，以及分区表在 HDFS 文件系统中的组织结构。

```
[hadoop-cs1:21000] > create database external_partitions;
Query: create database external_partitions

Returned 0 row(s) in 0.16s
```

为了将分区表的数据单独存放，我们可以创建一个单独的数据库。这里创建的数据库名称为 external_partitions。

```
[hadoop-cs1:21000] > use external_partitions;
Query: use external_partitions
```

如果我们想在新创建的数据库中创建对象，我们必须首先使用 USE 命令将当前数据库切换为 external_partitions。

```
[hadoop-cs1:21000] > create table logs (field1 string, field2 string, field3 string)
                   > partitioned by (year string, month string , day string, host
string)
                   > row format delimited fields terminated by ',';
Query: create table logs (field1 string, field2 string, field3 string) partitioned
by (year string, month string , day string, host string) row format delimited fields
terminated by ','
Returned 0 row(s) in 0.13s
```

这里我们创建了一张分区表，partitioned by (year string, month string , day string, host string)表示使用 year，month，day，host 作为分区列，row format delimited fields terminated by ','表示存储的数据文件以逗号分隔。另外，这里的分区列 year，month，day，host 都不出现在 create table 后面的对列定义的列表中，也就是这里的 logs (field1 string, field2 string, field3 string)部分中。

```
insert into logs partition (year="2013", month="07", day="28", host="host1")
values
    ("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="28", host="host2")
values
    ("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="29", host="host1")
values
    ("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="29", host="host2")
values
    ("foo","foo","foo");
insert into logs partition (year="2013", month="08", day="01", host="host1")
values
    ("foo","foo","foo");
[hadoop-cs1:21000] > insert into logs partition (year="2013", month="07", day="28",
host="host1") values
                  > ("foo","foo","foo");
Query: insert into logs partition (year="2013", month="07", day="28", host="host1")
values ("foo","foo","foo")
Inserted 1 rows in 2.40s
[hadoop-cs1:21000] > insert into logs partition (year="2013", month="07", day="28",
host="host2") values
                  > ("foo","foo","foo");
Query: insert into logs partition (year="2013", month="07", day="28", host="host2")
values ("foo","foo","foo")
Inserted 1 rows in 0.90s
[hadoop-cs1:21000] > insert into logs partition (year="2013", month="07", day="29",
host="host1") values
                  > ("foo","foo","foo");
Query: insert into logs partition (year="2013", month="07", day="29", host="host1")
values ("foo","foo","foo")
Inserted 1 rows in 1.00s
[hadoop-cs1:21000] > insert into logs partition (year="2013", month="07", day="29",
host="host2") values
                  > ("foo","foo","foo");
Query: insert into logs partition (year="2013", month="07", day="29", host="host2")
values ("foo","foo","foo")
Inserted 1 rows in 1.01s
```

```
[hadoop-cs1:21000] > insert into logs partition (year="2013", month="08", day="01", host="host1") values
                  > ("foo","foo","foo");
Query: insert into logs partition (year="2013", month="08", day="01", host="host1") values ("foo","foo","foo")
Inserted 1 rows in 0.81s
[hadoop-cs1:21000] >
```

我们使用 insert into 的形式向特定的分区中写入数据，这里写的数据都是"foo"、"foo"、"foo"。对于一个包含某些特定分区数据的单独创建的分区表，他们是以什么形式存储在 HDFS 上的呢？

```
[root@hadoop-cm1 ~]# su - hdfs
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse
Found 2 items
drwxrwxrwt   - impala hive          0 2014-11-18 13:07 /user/hive/warehouse/external_partitions.db
drwxrwxrwt   - impala hive          0 2014-11-18 12:46 /user/hive/warehouse/tab3
```

这里的文件夹为 external_partitions.db，external_partitions 是我们刚刚单独创建的数据库的名称，而.db 的后缀也是为了说明这是一个数据库对应的目录。

```
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/external_partitions.db
Found 1 items
drwxrwxrwt   - impala hive          0 2014-11-18 13:07 /user/hive/warehouse/external_partitions.db/logs
```

数据库目录 external_partitions.db 下的 logs 文件夹恰好是我们创建的分区表的名称。

```
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/external_partitions.db/logs
Found 2 items
drwxrwxrwt   - impala hive          0 2014-11-18 13:07 /user/hive/warehouse/external_partitions.db/logs/.impala_insert_staging
drwxr-xr-x   - impala hive          0 2014-11-18 13:07 /user/hive/warehouse/external_partitions.db/logs/year=2013
```

year=2013 是我们创建分区表是指定的 4 个分区列中的第一列。

```
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/external_partitions.db/logs/year=2013
Found 2 items
drwxr-xr-x   - impala hive          0 2014-11-18 13:07 /user/hive/warehouse/external_partitions.db/logs/year=2013/month=07
```

```
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=08
```

year=2013 目录下有 month=07 和 month=08。month 是我们创建分区表时指定的 4 个分区列中的第二列。

```
-bash-4.1$          hdfs          dfs          -ls
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07
Found 2 items
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=29
-bash-4.1$          hdfs          dfs          -ls
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=08
Found 1 items
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=08/day=01
```

month=07 和 month=08 的子目录下又有 day=28，day=29，day=01 的子目录，而 day 是我们创建分区表时指定的 4 个分区列中的第三列。

```
-bash-4.1$          hdfs          dfs          -ls
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28
Found 2 items
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=h
ost1
drwxr-xr-x   - impala hive          0 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=h
ost2
-bash-4.1$          hdfs          dfs          -ls
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=h
ost1
Found 1 items
-rw-r--r--   3 impala hive         12 2014-11-18 13:07
/user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=h
ost1/c9446623acb6363b-d2bbeef4df81f4af_1068880746_data.0
```

同样的，在 day=28 下又有 host=host1，host=host2 的子目录，而 host 是我们创建分区表时指定的 4 个分区列中的第四列。在 year=2013/month=07/day=28/host=host1 下，我们可以看到 Impala

自动生成的数据文件 c9446623acb6363b-d2bbeef4df81f4af_1068880746_data.0。

```
-bash-4.1$ hdfs dfs -cat /user/hive/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=host1/c9446623acb6363b-d2bbeef4df81f4af_1068880746_data.0
foo,foo,foo
-bash-4.1$
```

通过 hdfs 命令，我们可以查看到该数据文件以逗号分隔的形式存储了值 foo，foo，foo。

我们可以通过 drop table 删除该分区表。

```
[hadoop-cs1:21000] > drop table logs;
Query: drop table logs
```

分区表删除之后，对应的目录及文件也一并被级联删除。

```
[root@hadoop-cm1 ~]# su  - hdfs
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse
Found 2 items
drwxrwxrwt   -  impala  hive            0  2014-11-18  14:11 /user/hive/warehouse/external_partitions.db
drwxrwxrwt  - impala hive        0 2014-11-18 12:46 /user/hive/warehouse/tab3
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/external_partitions.db
-bash-4.1$
```

2.4 外部分区表

如果我们已经有了各分区对应的数据文件，只是希望这些文件以分区表的形式组织在一起，则可以使用外部分区表。与普通分区表类似，我们首先需要在 HDFS 上创建对应的目录结构，并将数据文件 put 到对应的位置，然后通过 CREATE TABLE 语句指向我们创建的 HDFS 目录就可以了。

```
-bash-4.1$ id
uid=494(hdfs) gid=490(hdfs) groups=490(hdfs),492(hadoop)
-bash-4.1$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host1
-bash-4.1$ hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host2
-bash-4.1$ hdfs dfs -mkdir -p
```

```
/user/impala/data/logs/year=2013/month=07/day=29/host=host1
    -bash-4.1$            hdfs            dfs            -mkdir            -p
/user/impala/data/logs/year=2013/month=08/day=01/host=host1
```

在 hdfs 用户下创建分区表的目录结构。由于每个分区对应分区列的具体值，这里我们创建四个分区作为示例。

```
    -bash-4.1$ echo "bar,baz,bletch">dummy_log_data
    -bash-4.1$ cat dummy_log_data
    bar,baz,bletch
```

在实际的生产环境中，每个分区对应的数据文件完全不同。但此处作为实验，我们为每个分区使用同一个数据文件。将该数据文件上传到各分区对应的目录中。

```
    -bash-4.1$ hdfs dfs -put dummy_log_data
    /user/impala/data/logs/year=2013/month=07/day=28/host=host1
    -bash-4.1$ /user/impala/data/logs/year=2013/month=07/day=28/host=host1
    -bash: /user/impala/data/logs/year=2013/month=07/day=28/host=host1: No such file
or directory
    -bash-4.1$          hdfs          dfs          -put          dummy_log_data
/user/impala/data/logs/year=2013/month=07/day=28/host=host1
    -bash-4.1$          hdfs          dfs          -put          dummy_log_data
/user/impala/data/logs/year=2013/month=07/day=28/host=host2
    -bash-4.1$          hdfs          dfs          -put          dummy_log_data
/user/impala/data/logs/year=2013/month=07/day=29/host=host1
    -bash-4.1$          hdfs          dfs          -put          dummy_log_data
/user/impala/data/logs/year=2013/month=08/day=01/host=host1
    -bash-4.1$
```

使用 CREATE EXTERNAL TABLE 语句创建外部分区表。通过 LOCATION 指向我们创建的目录对应的位置'/user/impala/data/logs'。

```
    [hadoop-cs1:21000] > use external_partitions;
    Query: use external_partitions
    [hadoop-cs1:21000] > create external table logs (field1 string, field2 string,
field3 string)
                   > partitioned by (year string, month string, day string, host
string)
                   > row format delimited fields terminated by ','
                   > location '/user/impala/data/logs';
    Query: create external table logs (field1 string, field2 string, field3 string)
```

```
partitioned by (year string, month string, day string, host string) row format delimited
fields terminated by ',' location '/user/impala/data/logs'

   Returned 0 row(s) in 0.17s
```

手动为该分区表添加分区，添加分区时需要指定该分区列的具体键值。

```
   [hadoop-cs1:21000] > alter table logs add partition
(year="2013",month="07",day="28",host="host1");
   Query: alter table logs add partition
(year="2013",month="07",day="28",host="host1")
   [hadoop-cs1:21000] > alter table logs add partition
(year="2013",month="07",day="28",host="host2");
   Query: alter table logs add partition
(year="2013",month="07",day="28",host="host2")
   [hadoop-cs1:21000] > alter table logs add partition
(year="2013",month="07",day="29",host="host1");
   Query: alter table logs add partition
(year="2013",month="07",day="29",host="host1")
   [hadoop-cs1:21000] > alter table logs add partition
(year="2013",month="08",day="01",host="host1");
   Query: alter table logs add partition
(year="2013",month="08",day="01",host="host1")
   [hadoop-cs1:21000] >
```

创建完成后即可对该分区表进行查询操作。

```
   [hadoop-cs1:21000] >
   [hadoop-cs1:21000] > select * from logs;
   Query: select * from logs
   +--------+--------+--------+------+-------+-----+-------+
   | field1 | field2 | field3 | year | month | day | host  |
   +--------+--------+--------+------+-------+-----+-------+
   | bar    | baz    | bletch | 2013 | 07    | 28  | host2 |
   | bar    | baz    | bletch | 2013 | 08    | 01  | host1 |
   | bar    | baz    | bletch | 2013 | 07    | 28  | host1 |
   | bar    | baz    | bletch | 2013 | 07    | 29  | host1 |
   +--------+--------+--------+------+-------+-----+-------+
   Returned 4 row(s) in 0.31s
```

删除之后，对应的数据文件的目录及数据文件仍然存在。其实对于外部表，通过 DROP TABLE

命令仅仅是删除标的定义而已。

```
[hadoop-cs1:21000] > drop table logs;
Query: drop table logs
通过 drop table 语句可以删除该分区表。
drwxr-xr-x   - hdfs impala          0 2014-11-18 14:09 /user/impala/data/logs
-bash-4.1$
```

2.5 笛卡尔连接

在数据仓库应用中，我们通常会通过笛卡尔乘积对两个或者多个不同的维度进行关联，以得到所有可能的维度组合。比如，通过年份和季度的组合，我们就可以得到所有年份的所有季度。

在 Impala 老版本中，如果想对两张表进行关联，必须至少要指定一个等值连接条件，否则就会报错。Impala 这样设计的初衷是对于两张没有关联条件的大表进行 JOIN，就会产生笛卡尔乘积，这样将生成一个巨大的结果集，消耗大量的集群资源。

我们创建了两张维度表，一张是年份，一张是季度。

```
[hadoop-cs1:21000] > create table dim_year (x int);
Query: create table dim_year (x int)

Returned 0 row(s) in 0.12s
[hadoop-cs1:21000] > insert into dim_year values(2012),(2013),(2014) ;
Query: insert into dim_year values(2012),(2013),(2014)
Inserted 3 rows in 1.74s
[hadoop-cs1:21000] > create table dim_quarter (y int);
Query: create table dim_quarter (y int)

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > insert into dim_quarter values(1),(2),(3),(4);
Query: insert into dim_quarter values(1),(2),(3),(4)
Inserted 4 rows in 1.60s
[hadoop-cs1:21000] >
```

如果这两张表没有等值的关联条件，Impala 会直接报错。

```
[hadoop-cs1:21000] > select x.x,y.y from dim_year x join dim_quarter y;
Query: select x.x,y.y from dim_year x join dim_quarter y
ERROR: NotImplementedException: Join with 'y' requires at least one conjunctive
```

```
equality predicate. To perform a Cartesian product between two tables, use a CROSS JOIN.
```

自 Impala 1.2.2 起，我们可以通过将 JOIN 替换为 CROSS JOIN 来显示的指定连接操作符进行笛卡尔连接操作。当然，一般情况下我们不会针对特别大的两张表进行笛卡尔操作。

```
[hadoop-cs1:21000] > select x.x,y.y from dim_year x cross join dim_quarter y;
Query: select x.x,y.y from dim_year x cross join dim_quarter y
+------+---+
| x    | y |
+------+---+
| 2012 | 1 |
| 2012 | 2 |
| 2012 | 3 |
| 2012 | 4 |
| 2013 | 1 |
| 2013 | 2 |
| 2013 | 3 |
| 2013 | 4 |
| 2014 | 1 |
| 2014 | 2 |
| 2014 | 3 |
| 2014 | 4 |
+------+---+
Returned 12 row(s) in 0.41s
```

通过制定 CROSS JOIN 作为查询操作符，可以正常进行笛卡尔连接操作。

2.6 更新元数据

本章示例中的 Impala 的数据文件我们都使用以逗号分隔的文本文件。事实上，Impala 可以对 RCFile、SequenceFile、Parquet、Avro 等多种文件格式进行查询。而对于某些类型的 Impala 支持的不太好的文件格式，Impala 只能对其进行查询操作，却无法向其中写入数据。

如果我们又不得不向其中写入数据，就必须通过 Hive 来进行。

对于通过 Hive 创建，删除，修改或者其他类型的操作，Impala 都无法自动识别到 Hive 中元数据的变更情况。如果想让 Impala 识别到这些变化，在连接到 impala-shell 后首先要做的操作就是 INVALIDATE METADATA，该语句将会使所有的 Impala 元数据失效并重新从元数据库同步元

数据信息。

```
-bash-4.1$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > invalidate metadata;
Query: invalidate metadata

Returned 0 row(s) in 1.25s
[hadoop-cs1:21000] >
```

对于通过 Hive 加载，插入，改变的数据操作，或者通过 hdfs 命令对数据文件进行的变更操作，Impala 都无法自动识别数据的变更情况。如果想让 Impala 识别到这些变化，在连接到 impala-shell 后，首先要做的操作就是 REFRESH table_name，该语句会让 Impala 识别到数据的变更情况。

```
-bash-4.1$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > refresh tab1;
Query: refresh tab1

Returned 0 row(s) in 0.29s
[hadoop-cs1:21000] >
```

第 3 章

Impala概念及架构

本章将主要介绍 Impala 的概念及技术架构，Impala 应用编程，以及与 Hadoop 集群生态系统的集成关系。通过对 Impala 概念及技术架构的阐述，将帮助我们从原理上初步理解 Impala 的运行原理及使用场景。通过对 Impala 应用编程的了解，我们可以针对不同的应用选择不同的方式调用 Impala。通过对 Impala 与其他 Hadoop 组件的集成关系，将帮助我们更好的将 Impala 与其他相关技术结合使用。

3.1 Impala 服务器组件

Impala 服务器是一个分布式，大规模并行处理（MPP）数据库引擎。它包括运行在 CDH 集群主机上的不同后台进程。

3.1.1 Impala Daemon

这个进程是运行在集群每个节点上的守护进程，是 Impala 的核心组件。在每个节点上这个进程的名称为 impalad。

```
   [root@hadoop-cs1 ~]# ps -ef|grep impalad
   impala          4712      1854      0    11:53    ?                     00:00:11
/usr/lib/impala/sbin-retail/impalad
--flagfile=/var/run/cloudera-scm-agent/process/146-impala-IMPALAD/impala-conf/imp
alad_flags

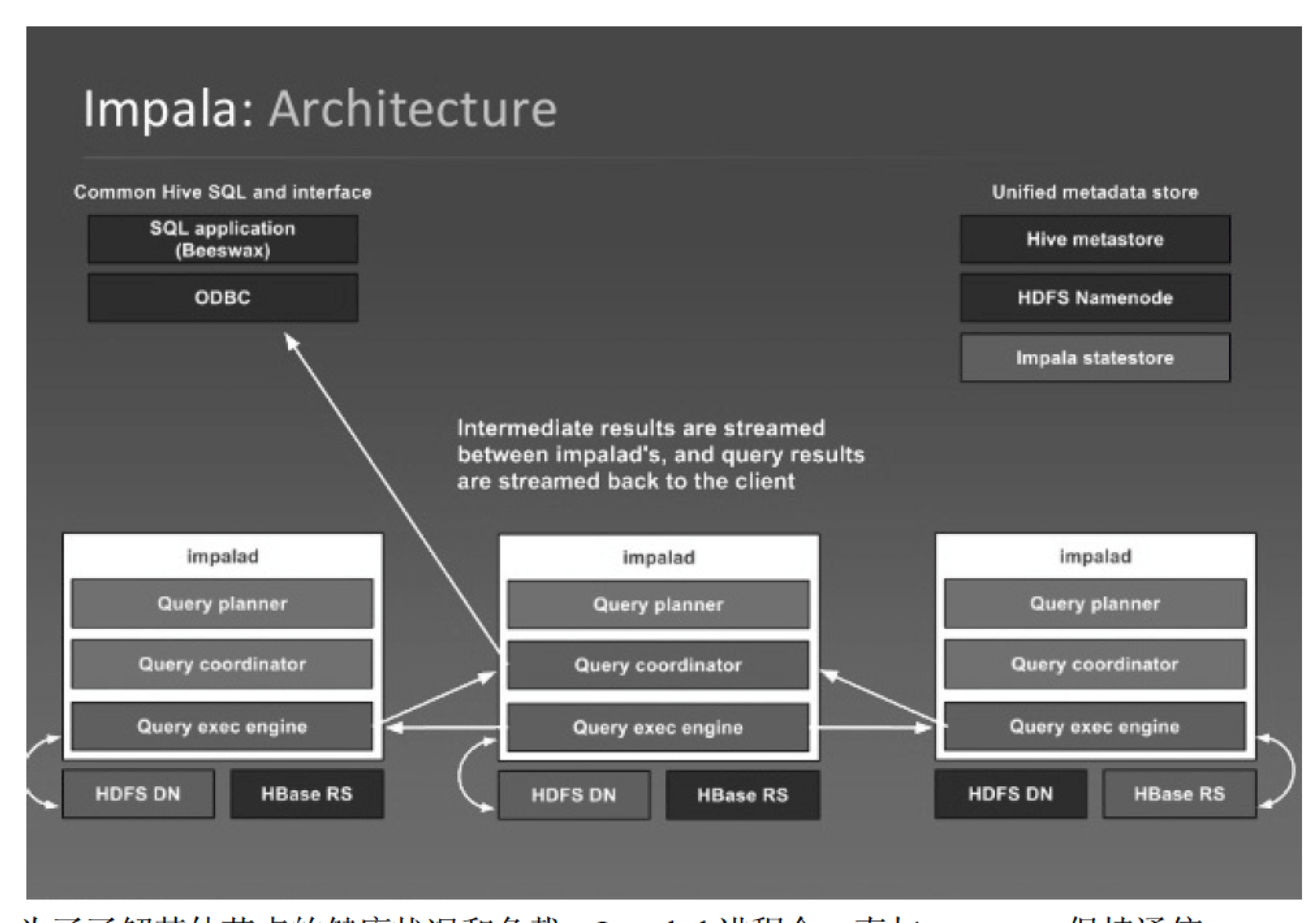
   [root@hadoop-cs2 ~]# ps -ef|grep impalad
   impala          4300      1835      0    11:53    ?                     00:00:12
/usr/lib/impala/sbin-retail/impalad
--flagfile=/var/run/cloudera-scm-agent/process/143-impala-IMPALAD/impala-conf/imp
alad_flags

   [root@hadoop-cs3 ~]# ps -ef|grep impalad
```

```
   impala           4330      1831      0   11:53   ?                   00:00:11
/usr/lib/impala/sbin-retail/impalad
--flagfile=/var/run/cloudera-scm-agent/process/145-impala-IMPALAD/impala-conf/imp
alad_flags
```

它负责读写数据文件；接受来自 impala-shell，Hue，JDBC 或 ODBC 的查询请求，与集群中的其他节点分布式并行工作，并将本节点的查询结果返回给中心协调者节点。

我们可以向任何一个节点的 impalad 进程提交一个查询，提交查询的这个节点将作为这个查询的“协调者节点”为这个查询提供服务。其他节点的运算结果会被传输到协调者节点，协调者节点将最终运算结果返回。当我们使用 impala-shell 进行功能性测试的时候，方便起见，我们总是会连接到同一个节点的 impalad 上。对于运行在生产环境上的 impala 集群来说，我们必须考虑到各节点的负载均衡，建议使用 JDBC/ODBC 接口以 round-robin 的方式将每个查询提交到不同节点的 impalad 进程上。Impala 架构如下图所示，读者可以到官网查阅相应的彩图。



为了了解其他节点的健康状况和负载，Impalad 进程会一直与 statestore 保持通信。

当 impala 集群中创建，修改或者删除了对象，或者进行了 INSERT/LOAD DATA 操作，catalogd 进程要向所有节点广播消息，以保证每个 impalad 节点都能够及时地了解整个集群中对象元数据的最新状态。后台进程间的通信最大限度的降低了对 REFRESH/INVALIDATE METADATA 命令的依赖。（但是对于和 impala 1.2 版本之前的节点通信，还是需要显示指定的。）

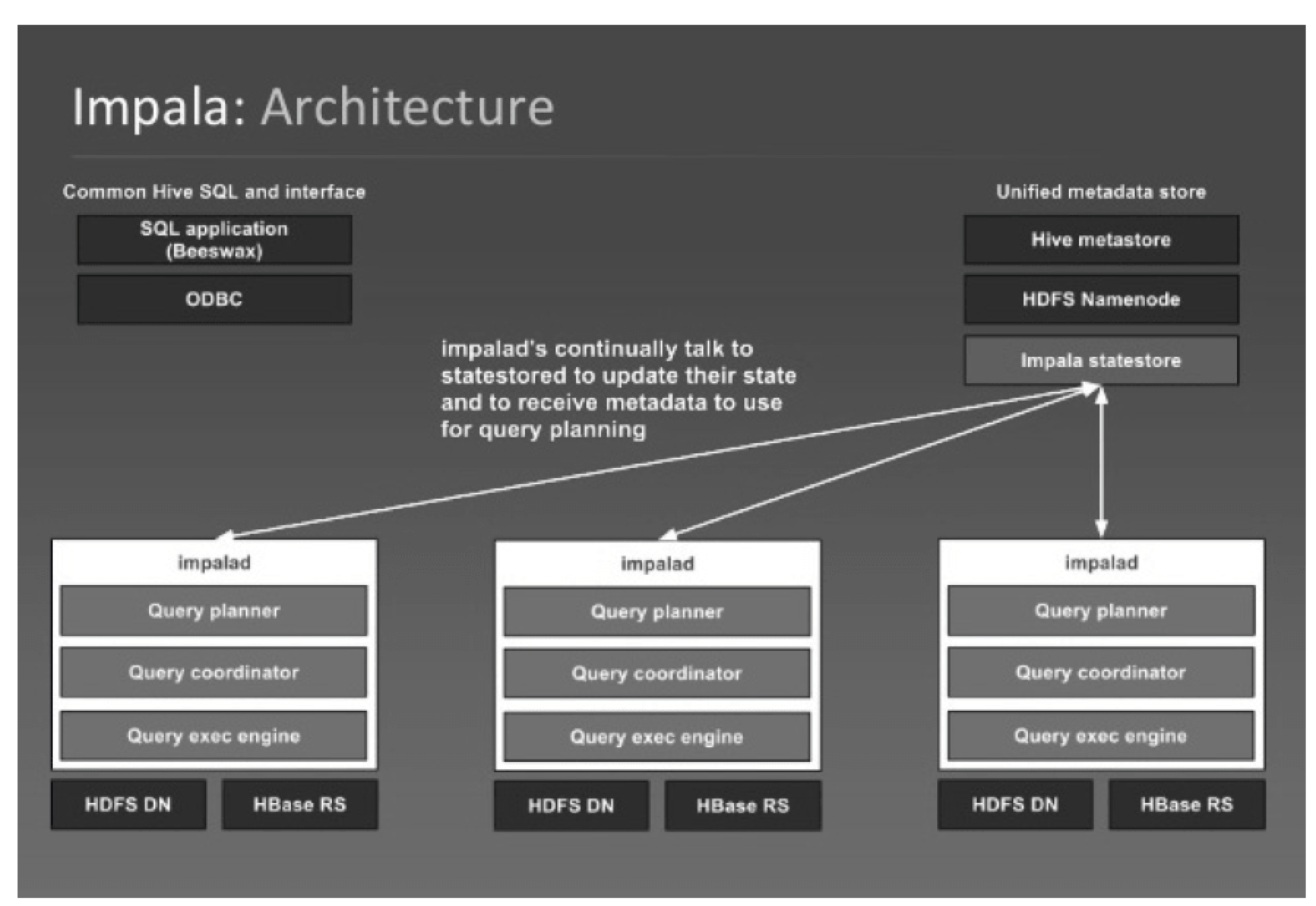
3.1.2 Impala Statestore

Statestore 搜集集群中 impalad 进程节点的健康状况，并不断地将健康状况的结果转发给所有

的 impalad 进程节点。Statestore 进程的名称为 statestored。

```
[root@hadoop-cm1 ~]# ps -ef|grep statestored
impala          5658      1844      0    11:53    ?                         00:00:08
/usr/lib/impala/sbin-retail/statestored
--flagfile=/var/run/cloudera-scm-agent/process/147-impala-STATESTORE/impala-conf/
state_store_flags
```

一个 impala 集群只需要一个 statestored 进程节点。当 impala 节点由于硬件故障，网络错误，软件问题或者其他原因导致节点不可用，statestore 将确保这一信息及时的传达到所有的 impalad 进程节点上，当有新的查询请求时，impalad 进程节点将不会把查询请求发送到不可用的节点上。如下图所示。



Statestore 的目的是在集群故障时对 impalad 进程节点同步信息，所以对于一个正常运行的 impala 集群来说，它并不是一个关键进程。如果 statestore 不可用，impalad 进程节点之间仍然可以相互协调正常对外提供分布式查询。在 statestore 不可用的情况下，impalad 进程节点失败，只是让集群不再那么强劲。当 statestore 恢复正常时，它将重新与 impalad 进程节点建立通信，恢复对集群的监控功能。

3.1.3 Impala Catalog

当 Impala 集群中执行的 SQL 语句会引起元数据变化时，catalog 服务负责将这些变化推送到其他 impalad 进程节点上。Catalog 服务对应的进程名称为 catalogd。一个 impala 集群只需要一个 catalogd 进程。因为所有的请求都是通过 statestored 进程发送过来，所以让 statestored 和 catalogd 运行在同一个节点上是一个不错的选择。

```
[root@hadoop-cm1 ~]# ps -ef|grep catalogd
impala      5630     1844     0   11:53   ?                  00:00:09
/usr/lib/impala/sbin-retail/catalogd
--flagfile=/var/run/cloudera-scm-agent/process/144-impala-CATALOGSERVER/impala-co
nf/catalogserver_flags
```

在 impala 1.2 版本中的这个新的进程大大减少了对 REFRESH/INVALIDATE METADATA 语句的元数据同步需求。通常情况下，如果在一个 impalad 进程节点上执行了 CREATE DATABASE,DROP DATABASE,CREATE TABLE,ALTER TABLE,DROP TABLE 语句，在其他 impalad 进程节点运行一个查询之前总是要执行 INVALIDATE METADATA 语句同步对象的元数据信息。同样的道理，如果在一个 impalad 进程节点执行 INSTERT 语句，其他节点在运行一个查询之前需要执行 REFRESH table_name，以便让这个节点知道这个表有新增的数据文件。当通过 impala 执行可能引起元数据变化的语句时，catalog 服务确保不必再执行 REFRESH/INVALIDATE METADATA 这样的元数据同步语句。但是如果是通过 Hive 进行创建表，加载数据等类似的操作时，还是需要在 impalad 进程节点上执行 REFRESH/INVALIDATE METADATA 操作的。

我们在某个 impalad 进程节点上执行了一个创建表和删除表的过程：

```
[hadoop-cs1:21000] > create table test(id int);
Query: create table test(id int)

Returned 0 row(s) in 2.07s
[hadoop-cs1:21000] > drop table test;
Query: drop table test
[hadoop-cs1:21000] >
```

随后我们在 catalogd 进程节点的日志中，可以看到以下信息：

```
I1120 12:43:49.726615   6972 CatalogOpExecutor.java:727] Creating table
default.test
I1120 12:43:49.726819    6972  MetaStoreClientPool.java:47]  Creating
MetaStoreClient. Pool Size = 0
I1120 12:43:49.727233 6972 HiveMetaStoreClient.java:257] Trying to connect to
metastore with URI thrift://hadoop-cm1:9083
I1120 12:43:49.731443 6972 HiveMetaStoreClient.java:345] Connected to metastore.
I1120 12:43:51.852354   5736  catalog-server.cc:292]  Publishing  update:
TABLE:default.test@13
I1120 12:43:51.852411   5736  catalog-server.cc:292]  Publishing  update:
CATALOG:50d44d9581344be7:9d70be84ccf1be02@13
I1120 12:43:52.352764  5745 catalog-server.cc:208] Catalog Version: 13 Last
Catalog Version: 12
```

```
I1120 12:45:27.077828 5722 TableLoadingMgr.java:224] Loading next table.
Remaining items in queue: 0
I1120 12:45:27.314470 7001 TableLoader.java:60] Loading metadata for:
default.test
I1120 12:45:27.477274 7001 MetaStoreClientPool.java:47] Creating
MetaStoreClient. Pool Size = 0
I1120 12:45:27.642734 7001 HiveMetaStoreClient.java:257] Trying to connect to
metastore with URI thrift://hadoop-cm1:9083
I1120 12:45:27.815253 7001 HiveMetaStoreClient.java:345] Connected to metastore.
I1120 12:45:28.095842 7001 HdfsTable.java:747] load table: default.test
I1120 12:45:28.949318 7001 HdfsTable.java:851] table #rows=-1
I1120 12:45:36.798162 5736 catalog-server.cc:292] Publishing update:
TABLE:default.test@14
I1120 12:45:41.634021 5736 catalog-server.cc:292] Publishing update:
CATALOG:50d44d9581344be7:9d70be84ccf1be02@14
I1120 12:45:57.034734 6972 CatalogOpExecutor.java:646] Dropping table/view
default.test
I1120 12:45:58.104498 6972 MetaStoreClientPool.java:47] Creating
MetaStoreClient. Pool Size = 0
I1120 12:45:58.305776 6972 HiveMetaStoreClient.java:257] Trying to connect to
metastore with URI thrift://hadoop-cm1:9083
I1120 12:45:58.511062 6972 HiveMetaStoreClient.java:345] Connected to metastore.
I1120 12:46:05.572391 5736 catalog-server.cc:292] Publishing update:
CATALOG:50d44d9581344be7:9d70be84ccf1be02@15
I1120 12:46:05.572418 5736 catalog-server.cc:311] Publishing deletion:
TABLE:default.test
I1120 12:47:04.802428 5745 catalog-server.cc:208] Catalog Version: 15 Last
Catalog Version: 15
I1120 12:49:34.963711 5745 catalog-server.cc:208] Catalog Version: 15 Last
Catalog Version: 15
```

可以发现在创建和删除表的过程中，catalogd 进程负责连接到元数据库并进行元数据更新操作。

3.2 Impala 应用编程

Impala 核心的开发语言是 SQL 语句。Impala 也可以通过 JDBC/ODBC 接口为其他语言提供服

务。许多 BI 工具就是通过 JDBC/ODBC 对 Impala 进行调用的。对于某种特定类型的分析需求，我们也可以使用 C++或者 Java 编写 SQL 内嵌函数 UDF。

3.2.1 Impala SQL 方言

Impala SQL 方言继承了 Apache HiveQL 的 SQL 语法。对于已经有在 Hadoop 上有 SQL 使用经验的使用者来说，转向 Impala SQL 将非常容易。目前，Impala SQL 支持 HiveQL 语句，数据类型和内嵌函数的子集。

对于曾经有传统的数据库经验的使用者来说，有一下几点需要注意：

（1）Impala SQL 的重点在于查询，所以只包含很少的 DML 语句。它不具备 UPDATA/DELETE 语句。对于过期的数据通常使用直接删除（通过 DROP TABLE,ALTER TABLE...DROP PARTITION 语句）或者替换（INSERT OVERWRITE 语句）的方式变相删除。

（2）数据加载通过 INSERT 语句完成，通常是通过对其他表的查询转换后进行批量插入操作。可以使用 INSERT INTO 向已存在的数据的表添加数据，也可以使用 INSERT OVERWRITE 语句将整个表或者分区的内容替换，但是没有针对单行记录操作的 INSERT … VALUES 语法。

（3）我们常常需要使用 Hadoop 现有的数据文件创建 Impala 表定义，然后使用 Impala 进行实时查询。这些数据文件和表元数据可以被 Hadoop 生态系统其他组件共享。

（4）Imapla 适用于数据仓库类型大数据集进行操作，Impala SQL 支持类似于传统数据库的数据加载方式。我们可以基于一个逗号分隔的文本文件使用 CREATE TABLE 语句创建一张外部表。外部表可以从现有的数据文件中读取数据，而不需要移动或者转换数据文件。

（5）因为 Impala 读取的大量数据可能不是完全对齐的，它不支持字符串类型的长度限制。我们可以定义 String 作为一个数据列，而不是 CHAR(1)或 VARCHAR(64)。

（6）对于查询密集型应用程序，你会发现跟传统数据库类似的概念，比如：连接，处理字符串，数字，日期的内嵌函数，聚合函数，子查询和比较操作符等。

（7）在数据仓库中，我们将经常使用分区表。

（8）在 Impala 1.2 或者更高版本中，我们可以通过 UDFs、执行自定义的比较和转换逻辑。

3.2.2 Impala 编程接口概述

可以通过如下方式向 Impala 提交请求：

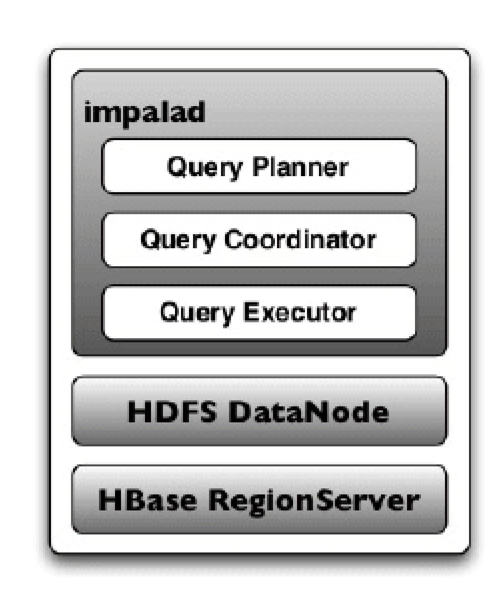
- impala-shell 交互式命令行
- Apache Hue 基于 web 的用户接口
- JDBC
- ODBC

通过以上各种方式，我们可以在异构环境中使用 Impala，通过 JDBC 或 ODBC 将应用运行在非 Linux 平台上，也可以通过 JDBC 或 ODBC 接口将 Impala 集成在不同的 BI 工具中。

每个 impalad 进程运行在集群中不同的物理节点上，通过端口监听发送过来的请求。来自 impala-shell 和 Hue 的请求将被路由到 impalad 进程的同一端口。Impalad 进程也通过不同的端口监听 JDBC 或 ODBC 请求。

3.3 与 Hadoop 生态系统集成

Impala 可以使用 Hadoop 生态系统内的许多熟悉的组件。Impala 可以与其他 Hadoop 组件交换数据，它既可以作为生产者也可以作为消费者，以非常灵活的方式在 ETL 或 ELT 过程中使用。



3.3.1　与 Hive 集成

Impala 一个主要的目标就是让 Hadoop 上的 SQL 操作更加快速，高效到能够吸引其他在 Hadoop 上运行 SQL 的用户。比如：它就是对于在 Hadoop 上运行 Hive 执行长时间运行，面向批处理 SQL 查询的用户的福音。

Impala 使用传统的 MYSQL 或 Postgresql 作为元数据库存储表定义信息，Hive 也是使用同样的方式保存元数据。只要 Hive 使用的是 Impala 支持的数据类型，文件格式或者压缩方式，Impala 都可以直接访问。这意味着 impala 可以使用 SELECT 读取不同类型的数据，然后使用 INSERT 语句进行数据加载。Impala 可支持的数据类型包括 Avro、RCFile 或 SequeceFile 等格式，当然我们也可以使用 Hive 加载这些类型的数据。

Impala 查询优化器可使用表统计信息和列统计信息。在 Hive 中，我们可以通过 ANALYZE TABLE 语句搜集这些信息。在 Impala 1.2.2 或者更高版本中，可以使用 COMPUTE STATS 语句替换上述语句。COMPUTE STATS 需要更少的设置，更可靠，更快，而且不需要再 impala-shell 和 Hive shell 之前切换的开销。

3.3.2　与 HDFS 集成

Impala 使用分布式的文件系统 HDFS 作为主要的数据存储方式。Impala 依赖于 HDFS 的冗余

机制来避免节点的硬件或者网络故障。Impala 表的数据以文件的形式存储在 HDFS 中，存储时我们可以使用熟悉的 HDFS 文件格式和压缩格式。对于一个新的表，Impala 将读取表定义中指定的目录中的所有文件作为数据。当由 Impala 向表中添加的数据时，在相应的目录中 Impala 将自动生成新的数据文件。

3.3.3 使用 HBase

HBase 是替代 HDFS 作为 Impala 的数据存储的另一种方式。它是构建与 HDFS 顶层的一个数据库存储系统，但是它不支持 SQL 语句。许多 Hadoop 使用者使用 HBase 作为海量数据存储使用。通过在 Impala 中定义到 HBase 表的映射关系，我们可以实现通过 Impala 查询 HBase 中的数据，甚至可以实现 Impala 和 HBase 表的连接查询。

第 4 章

SQL语句

Cloudera Impala 使用 SQL 作为查询语言。为了最大限度上防止用户在开发技能上的重复投入，Impala 保持了与 HiveQL 高度兼容性：

- 因为 Impala 与 Hive 使用同样的元数据记录关于表结构和属性的信息，Impala 可以访问通过 Impala 的 CREATE TABLE 语句创建的表，也可以访问使用 Hive DDL 定义的表。
- Impala 支持与 HiveQL 类似的 DML 语句。
- Impala 提供很多与 HiveQL 具有相同名称和参数类型的内嵌函数。

Impala 支持大多数的 HiveQL 的语句和语法，包括但不限于 JOIN、AGGREGATE、DISTINCT、UNION ALL、ORDER BY 、LIMIT 及非关联子查询等。Impala 也支持 INSERT INTO 和 INSERT OVERWRITE 语句。

Impala 支持与 Hive 数据类型完全相同名称和语义的数据类型，包括：STRING、TINYINT、SMALLINT、INT、BIGINT、FLOAT、DOUBLE、BOOLEAN、TIMESTAMP。大多数的 HiveQL 的 SELECT 和 INSERT 语句可以不做修改在 Impala 上直接运行。

4.1 注释

Impala 有两种格式的 SQL 注释方法。

1. 注释符号“-”

该符号到行尾所有的内容将被作为注释忽略。这种类型的注释只对单行有效。注释可以独立成行，或者作为 SQL 语句的一部分。

```
[hadoop-cs1:21000] > --这是一个注释
                   > SELECT VERSION();
Query: SELECT VERSION()
+---------------------------------------------+
| version()                                   |
+---------------------------------------------+
| impalad version 1.3.1-cdh5 RELEASE (build ) |
| Built on Mon, 09 Jun 2014 09:30:26 PST      |
```

```
+--------------------------------------------+
Returned 1 row(s) in 0.50s
```

2. 注释符号 "/* ...*/"

在 "/* ...*/" 内部的内容将被作为注释忽略。这种类型的注释可以跨多行生效。注释可以单独占用一行或多行，可在 SQL 语句任何位置出现。

```
[hadoop-cs1:21000] > /*这是一个注释*/
                   > SELECT VERSION();
Query: SELECT VERSION()
+--------------------------------------------+
| version()                                  |
+--------------------------------------------+
| impalad version 1.3.1-cdh5 RELEASE (build ) |
| Built on Mon, 09 Jun 2014 09:30:26 PST      |
+--------------------------------------------+
Returned 1 row(s) in 0.08s
```

4.2 数据类型

4.2.1 BIGINT

8 字节的整形数据类型，可以在 CREATE TABLE 和 ALTER TABLE 语句中使用。

范围：-9223372036854775808~9223372036854775807，没有 UNSIGNED 子类型。

转换：Impala 自动转换为单精度或双精度浮点类型。使用 CAST()可以转换成 TINYINIT、SMALLINT、INT、STRING 或 TIMESTAMP。 将整数值 N 转换成 TIMESTAMP 将会产生一个时间戳，该时间戳记录了从 1970 年 1 月 1 号开始经过 N 秒之后的时间。

```
[hadoop-cs1:21000] > select cast(now() as bigint);
Query: select cast(now() as bigint)
+-----------------------+
| cast(now() as bigint) |
+-----------------------+
| 1416490083            |
+-----------------------+
Returned 1 row(s) in 0.10s
```

```
[hadoop-cs1:21000] > select cast(1416490083 as timestamp);
Query: select cast(1416490083 as timestamp)
+-------------------------------+
| cast(1416490083 as timestamp) |
+-------------------------------+
| 2014-11-20 13:28:03           |
+-------------------------------+
Returned 1 row(s) in 0.13s
```

4.2.2　BOOLEAN

布尔数据类型可以使用在 CREATE TABLE 和 ALTER TABLE 语句中。可以为 TRUE 或者 FALSE。

范围：TRUE 或 FALSE。在 TRUE 或 FALSE 外面不要使用引号。我们可以将 TRUE 或 FALSE 写成大写、小写，或者大小写混合。从表中查询返回的布尔型总是返回小写的 true 或者 false。

转换： Impala 不会自动将其他任何类型转换为布尔型。我们可以使用 CAST()强制将一个整数或者浮点数转换成布尔型。值 0 被转换为 false，其他任何非 0 值被转换为 true。我们不能将一个字符串类型转换为布尔型。即使我们可以将布尔型强制转换为 String 类型，对于 true 值将返回字符串‘1’，对于 false 值将返回字符串‘0’。

```
[hadoop-cs1:21000] > select cast(0 as boolean);
Query: select cast(0 as boolean)
+--------------------+
| cast(0 as boolean) |
+--------------------+
| false              |
+--------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > select cast(1 as boolean);
Query: select cast(1 as boolean)
+--------------------+
| cast(1 as boolean) |
+--------------------+
| true               |
+--------------------+
Returned 1 row(s) in 0.19s
[hadoop-cs1:21000] > select cast(3 as boolean);
Query: select cast(3 as boolean)
+--------------------+
```

```
| cast(3 as boolean) |
+--------------------+
| true               |
+--------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select cast(-1 as boolean);
Query: select cast(-1 as boolean)
+---------------------+
| cast(-1 as boolean) |
+---------------------+
| true                |
+---------------------+
Returned 1 row(s) in 0.09s
```

4.2.3 DOUBLE

8 字节的双精度浮点类型，可以被使用在 CREATE TABLE 和 ALTER TABLE 中。

范围：4.94065645841246544e-324 ~1.79769313486231570e+308，正数或负数。

转换：Impala 不会自动将双精度浮点数转换成任何类型。但是我们可以使用 CAST()将双精度浮点型转换为 FLOAT、TINYINT、SMALLINT、INT、BIGINT、STRING、TIMESTAMP 或 BOOLEAN。我们可以使用对数计数法将 DOUBLE 型转换成 STRING 型。例如：可以用 1.0e6 代表一百万。

DOUBLE 还有一个别名叫做 REAL。

```
[hadoop-cs1:21000] > select cast(4.9e-324  as int);
Query: select cast(4.9e-324  as int)
+-----------------------+
| cast(4.9e-324 as int) |
+-----------------------+
| 0                     |
+-----------------------+
Returned 1 row(s) in 0.11s

[hadoop-cs1:21000] > select cast(5.9e-300  as string);
Query: select cast(5.9e-300  as string)
+--------------------------+
| cast(5.9e-300 as string) |
+--------------------------+
| 5.8999999999999998e-300  |
+--------------------------+
```

```
Returned 1 row(s) in 0.10s
```

4.2.4　FLOAT

4 字节单精度浮点型，可以使用在 CREATE TABLE 和 ALTER TABLE 中。

范围：1.40129846432481707e-45 .. 3.40282346638528860e+38，正数或者负数。

转换：Impala 自动转换 FLOAT 为精度更好的 DOUBLE 型，但是不会被自动转换为其他任何类型。我们可以使用 CAST()将 FLOAT 型转换为 TINYINT、SMALLINT、INT、BIGINT、STRING、TIMESTAMP 或 BOOLEAN。我们可以使用对数计数法将 FLOAT 型转换成 STRING 型。例如：可以用 1.0e6 代表一百万。

```
[hadoop-cs1:21000] > select cast(1.4e-45  as int);
Query: select cast(1.4e-45  as int)
+----------------------+
| cast(1.4e-45 as int) |
+----------------------+
| 0                    |
+----------------------+
Returned 1 row(s) in 0.09s

[hadoop-cs1:21000] > select cast(1.4e-45  as string);
Query: select cast(1.4e-45  as string)
+-------------------------+
| cast(1.4e-45 as string) |
+-------------------------+
| 1.40000000000000001e-45 |
+-------------------------+
Returned 1 row(s) in 0.20s
```

4.2.5　INT

4 字节整形数据类型，可以被使用在 CREATE TABLE 或 ALTER TABLE 语句中。

范围：-2147483648 .. 2147483647，没有 UNSIGNED 子类型。

转换：Impala 自动转换一个 INT 型无法表示的更大的整数位 BIGINT 型，FLOAT 型或者 DOUBLE 型。我们可以使用 CAST()将 INT 型转换为 TINYINT、SMALLINT、STRING、TIMESTAMP 型。将整数值 N 转换成 TIMESTAMP 将会产生一个时间戳，该时间戳记录了从 1970 年 1 月 1 号开始经过 N 秒之后的时间。

INT 类型的别名为 INTEGER。

```
[hadoop-cs1:21000] > select cast(now() as int);
```

```
Query: select cast(now() as int)
+--------------------+
| cast(now() as int) |
+--------------------+
| 1416490882         |
+--------------------+
Returned 1 row(s) in 0.10s
[hadoop-cs1:21000] > select cast(1416490882 as timestamp);
Query: select cast(1416490882 as timestamp)
+------------------------------+
| cast(1416490882 as timestamp) |
+------------------------------+
| 2014-11-20 13:41:22          |
+------------------------------+
Returned 1 row(s) in 0.09s
```

4.2.6 REAL

REAL 是 DOUBLE 类型的别名。

以下示例显示我们可以分别使用 REAL 或 DOUBLE 型，但是最终 Impala 会将它们统一作为 DOUBLE 型对待：

```
[hadoop-cs1:21000] > create table r1 (x real);
Query: create table r1 (x real)

Returned 0 row(s) in 0.22s
[hadoop-cs1:21000] > desc r1;
Query: describe r1
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | double |         |
+------+--------+---------+
Returned 1 row(s) in 0.88s
```

4.2.7 SMALLINT

2 字节的整型数据类型，可以在 CREATE TABLE 和 ALTER TABLE 语句中使用。

范围：-32768 .. 32767，无 UNSIGNED 子类型。

转换：Impala 自动转换为 INT、BIGINT 或浮点类型。使用 CAST()可以转换为 TINYINT、

STRING、TIMESTAMP。将整数值 N 转换成 TIMESTAMP 将会产生一个时间戳，该时间戳记录了从 1970 年 1 月 1 号开始经过 N 秒之后的时间。

```
[hadoop-cs1:21000] > select cast(32767 as timestamp);
Query: select cast(32767 as timestamp)
+--------------------------+
| cast(32767 as timestamp) |
+--------------------------+
| 1970-01-01 09:06:07      |
+--------------------------+
Returned 1 row(s) in 0.09s
```

4.2.8　STRING

该数据类型可以被用于 CREATE TABLE 和 ALTER TABLE 语句中。

长度：32767 字节。当声明 STRING 列时，不能像声明 VARCHAR、CHAR 或其他关系型数据库列类型一样使用长度限制。

字符集：为了实现对 Impala 子系统的完全支持，需要限制 STRING 的值为 ASCII 字符集。UTF-8 字符数据可以被 Impala 存储，也可以通过查询正常获取数据。但是由于 UTF-8 字符串包含非 ASCII 字符，所以不能保证它可以和字符串操作函数，比较运算符或者 ORDER BY 语句正常工作。

对于像 ISO-8859-1 或 ISO-8859-2 这种扩展了 ASCII 字符集的国家字符集，Impala 在表定义中无法包含这些字符集元数据。如果我们需要针对这些使用了国家字符集的字符数据进行排序，操作或者显示，我们只能在应用端自己实现。

转换：Impala 不会自动将 STRING 数据转换成任何数字类型。如果 STRING 数据使用了与 TIMESTAMP 匹配的格式，Impala 可以自动将 STRING 转换为 TIMESTAMP 类型。

我们可以使用 CAST()将 STRING 类型转换为 TINYINT、SMALLINT、INT、BIGINT、FLOAT、DOUBLE 或 TIMESTAMP。

我们不能直接将 STRING 类型强制转换为 BOOLEAN 类型，但是可是使用 CASE 表达式针对不同的 STRING 数据返回 TRUE 或 FALSE。

我们可以强制将 BOOLEAN 类型转换为 STRING 类型。对于 TRUE 返回 STRING ‘1’，对于 FLASE 返回 STRING ‘0’。

```
[hadoop-cs1:21000] > select cast("1123" as int);
Query: select cast("1123" as int)
+---------------------+
| cast('1123' as int) |
+---------------------+
| 1123                |
```

```
+---------------------+
Returned 1 row(s) in 0.09s

[hadoop-cs1:21000] > select cast(false as int);
Query: select cast(false as int)
+--------------------+
| cast(false as int) |
+--------------------+
| 0                 |
+--------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select cast(true as int);
Query: select cast(true as int)
+-------------------+
| cast(true as int) |
+-------------------+
| 1                |
+-------------------+
Returned 1 row(s) in 0.09s
```

4.2.9 TIMESTAMP

TIMESTAMP 它可以被使用在 CREATE TABLE 或 ALTER TABLE 语句中，表示某一个时间点。

范围：在底层，时间戳的精度为纳秒。

INTERVAL 表达式：

通过使用 INTERVAL 关键字，+ - 操作符 或者 date_add() date_sub()函数，我们可以实现针对日期型数据某个时间单元的数学运算。我们可以指定的时间单元包括: YEAR[S]、MONTH[S]、WEEK[S]、DAY[S]、HOUR[S]、MINUTE[S]、SECOND[S]、MILLISECOND[S]、MICROSECOND[S] 和 NANOSECOND[S].我们可以在 INTERVAL 表达式中指定某一个时间单元，比如： INTERVAL 3 DAYS 或 INTERVAL 25 HOURS，也可以将不同的时间单元组合在一起使用，比如：timestamp_value + INTERVAL 3 WEEKS - INTERVAL 1 DAY + INTERVAL 10 MICROSECONDS。

示例：

```
[hadoop-cs1:21000] > select now() + interval 1 day;
Query: select now() + interval 1 day
+-------------------------------+
| now() + interval 1 day        |
```

```
+-----------------------------+
| 2014-11-21 13:49:54.889215000 |
+-----------------------------+
Returned 1 row(s) in 0.10s
[hadoop-cs1:21000] > select date_sub(now(), interval 5 minutes);
Query: select date_sub(now(), interval 5 minutes)
+-----------------------------------+
| date_sub(now(), interval 5 minutes) |
+-----------------------------------+
| 2014-11-20 13:45:17.388729000      |
+-----------------------------------+
Returned 1 row(s) in 0.12s
```

时区：

为了避免无法预期的结果，Impala 使用 GMT 时间，而不使用本地时区存储时间戳。

转换：

Impala 可以自动将格式匹配的 STRING 类型转换为 TIMESTAMP 类型。TIMESTAMP 可以接受的格式为 YYYY-MM-DD HH:MM:SS.sssssssss ，其中可以只包含日期，或者只包含时间，秒后面的精度为可选项。比如，我们可以指定一个时间戳为'1966-07-30'，'08:30:00'，或 '1985-09-25 17:45:30.005'。我们也可以强制将一个整数或者浮点数 N 转换为 TIMESTAMP 类型，它代表自 1970 年 1 月 1 号经过 N 秒后对应的时间。

分区：

我们无法使用 TIMESTAMP 列作为一个分区键，但是可以从 TIMESTAMP 中抽取年、月、日、时等具体的值作为分区键，因为分区列的值将在 HDFS 上作为目录名称，而不是包含在数据文件中的字段。通过以上方式，我们仍然可以保存原始的 TIMESTAMP 值，而不必额外的复制一份数据浪费存储空间。

示例：

```
[hadoop-cs1:21000] > select cast('1966-07-30' as timestamp);
Query: select cast('1966-07-30' as timestamp)
+-------------------------------+
| cast('1966-07-30' as timestamp) |
+-------------------------------+
| 1966-07-30 00:00:00           |
+-------------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select cast('1985-09-25 17:45:30.005' as timestamp);
```

```
Query: select cast('1985-09-25 17:45:30.005' as timestamp)
+-------------------------------------------+
| cast('1985-09-25 17:45:30.005' as timestamp) |
+-------------------------------------------+
| 1985-09-25 17:45:30.005000000             |
+-------------------------------------------+
Returned 1 row(s) in 0.19s
[hadoop-cs1:21000] > select cast('08:30:00' as timestamp);
Query: select cast('08:30:00' as timestamp)
+------------------------------+
| cast('08:30:00' as timestamp) |
+------------------------------+
| 08:30:00                     |
+------------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select hour('1970-01-01 15:30:00');
Query: select hour('1970-01-01 15:30:00')
+----------------------------+
| hour('1970-01-01 15:30:00') |
+----------------------------+
| 15                         |
+----------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select hour('1970-01-01 15:30');
Query: select hour('1970-01-01 15:30')
+-------------------------+
| hour('1970-01-01 15:30') |
+-------------------------+
| NULL                    |
+-------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select hour('1970-01-01 27:30:00');
Query: select hour('1970-01-01 27:30:00')
+----------------------------+
| hour('1970-01-01 27:30:00') |
+----------------------------+
| NULL                       |
+----------------------------+
Returned 1 row(s) in 0.19s
```

```
[hadoop-cs1:21000] > select dayofweek('2004-06-13');
Query: select dayofweek('2004-06-13')
+-------------------------+
| dayofweek('2004-06-13') |
+-------------------------+
| 1                       |
+-------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select dayname('2004-06-13');
Query: select dayname('2004-06-13')
+-----------------------+
| dayname('2004-06-13') |
+-----------------------+
| Sunday                |
+-----------------------+
Returned 1 row(s) in 0.19s
[hadoop-cs1:21000] > select date_add('2004-06-13', 365);
Query: select date_add('2004-06-13', 365)
+-----------------------------+
| date_add('2004-06-13', 365) |
+-----------------------------+
| 2005-06-13 00:00:00         |
+-----------------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select day('2004-06-13');
Query: select day('2004-06-13')
+-------------------+
| day('2004-06-13') |
+-------------------+
| 13                |
+-------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select datediff('1989-12-31','1984-09-01');
Query: select datediff('1989-12-31','1984-09-01')
+-------------------------------------+
| datediff('1989-12-31', '1984-09-01') |
+-------------------------------------+
| 1947                                |
+-------------------------------------+
```

```
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select now();
Query: select now()
+-------------------------------+
| now()                         |
+-------------------------------+
| 2014-11-20 13:54:11.704774000 |
+-------------------------------+
Returned 1 row(s) in 0.20s
```

4.2.10 TINYINT

1 字节的整型数据类型，可以使用在 CREATE TABLE 或 ALTER TABLE 语句中。

范围： -128~127 ，没有 UNSIGNED 子类型。

转换：Impala 自动将一个大的整型数据转换为 SMALLINT、INT、BIGINT 或 浮点型（FLOAT 或 DOUBLE）。我们可以使用 CAST()将 TINYINT 类型强制转换为 STRING 类型或 TIMESTAMP 类型。将整数值 N 转换成 TIMESTAMP 将会产生一个时间戳，该时间戳记录了从 1970 年 1 月 1 号开始经过 N 秒之后的时间。

```
[hadoop-cs1:21000] > select cast(127 as timestamp);
Query: select cast(127 as timestamp)
+------------------------+
| cast(127 as timestamp) |
+------------------------+
| 1970-01-01 00:02:07    |
+------------------------+
Returned 1 row(s) in 0.10s
```

4.3 常量

Impala 中每种数据类型都有标记表示常量的方法。我们可以在像 SELECT 后面的选择列表、WHERE 子查询或者函数调用中的参数等类似的 SQL 中使用常量。

4.3.1 数值常量

为了表示整型数据类型（TINYINT、SMALLINT、INT、BIGINT）的常量，我们可以使用一个数字的序列表示。这个数字序列的首位可以为 0，但是首位的 0 总是会被忽略。

为了表示浮点型数据类型（FLOAT、DOUBLE），我们可以使用包含数点‘.’的数字序列。根据常量在 SQL 的上下文环境，整型常量可以升级为浮点型常量。对于指数常量，我们可以使用包含字符‘e’的数字序列表示。例如：1e6 表示的是 10 的 6 次方，也就是 1000000。一个指数的常量默认会被 Impala 解释为浮点型。

4.3.2　字符串常量

字符串常量使用单引号或者双引号引用。为了在字符常量中表示单引号，我们需要用双引号来引用这段字符。同样的，为了在字符常量中表示双引号，我们需要用单引号来引用这段字符。

为了在字符串常量中表示特殊字符，需要在转义字符前使用反斜杠“\”：

- \t 表示制表符。
- \n 表示换行。在 impala-shell 中输出时会另起一行。
- \r 表示回车（从当前位置移至本行开头）。在 impala-shell 中输出时可能会由于某些内容被覆盖而使格式混乱。
- \b 表示退格。在 impala-shell 中输出时可能会由于某些内容被覆盖而使格式混乱。
- \0 表示 ASCII 码的 nul 字符（不同于 SQL 中的 NULL）。在 impala-shell 中输出时不可见。
- \Z 表示 DOS 文件结束符。在 impala-shell 中输出时不可见。
- \%、_ 字符串经 LIKE 操作符进行类似查找时用来做转义通配符。
- \ 后面用三个八进制数字，代表一个 ASCII 码的字符。例如：\101 表示的是 ASCII 码的 65，对应的字符就是‘A’。
- \\ 两个反斜杠阻止转义字符进行转义。

如果在使用引号引用的字符串常量内包含引号字符，需要使用反斜杠进行转义。

如果在\后面的字符不是上述的转义字符，那么字符将不会被转义，仅代表该字符本身。

4.3.3　布尔常量

对于布尔值，常量为 TRUE 或者 FALSE，无须使用引号引用，大小写均可。

示例：

```
[hadoop-cs1:21000] > select true;
Query: select true
+------+
| true |
+------+
| true |
+------+
Returned 1 row(s) in 0.09s
```

```
[hadoop-cs1:21000] > create table test(id int,ismale boolean);
Query: create table test(id int,ismale boolean)

Returned 0 row(s) in 0.14s
[hadoop-cs1:21000] > insert into test values(1,true);
Query: insert into test values(1,true)
Inserted 1 rows in 0.94s
[hadoop-cs1:21000] > select * from test where ismale=true;
Query: select * from test where ismale=true
+----+--------+
| id | ismale |
+----+--------+
| 1  | true   |
+----+--------+
Returned 1 row(s) in 0.68s
[hadoop-cs1:21000] >
```

4.3.4 时间戳常量

Impala 可以自动把格式正确的 STRING 常量转换为 TIMESTAMP 常量。TIMESTAMP 可以接受的格式为 YYYY-MM-DD HH:MM:SS.sssssssss ，其中可以只包含日期，或者只包含时间，秒后面的精度为可选项。比如，我们可以指定一个时间戳为'1966-07-30'、 '08:30:00'， 或 '1985-09-25 17:45:30.005'。我们也可以强制将一个整数或者浮点数 N 转换为 TIMESTAMP 类型，它代表自 1970 年 1 月 1 号经过 N 秒后对应的时间。

4.3.5 NULL

Impala 中的 NULL 值与其他的数据库系统的类似，但不同的数据库系统对 NULL 的限制和行为又有所不同。对于大数据处理来说，NULL 值得语义准备异常重要。任何歧义都可能导致结果不准确或者数据格式错误，对于大数据集的结果校准或者数据格式纠正都是相当耗费时间的事情。

（1）NULL 值不同于空字符串。空字符串是""或者''，代表的是在单引号或者双引号中间什么也没有的字符串常量。

（2）在一个分隔符分隔的文本文件中，NULL 值被表示为 \N。

（3）如果我们使用 Impala 向分区表插入数据时，分区列为 NULL 或者空字符串，数据将会被写到一个特殊的分区里。这个分区专门用于存储列为 NULL 的值或者空字符串。当我们对这些值进行查询的时候，无论原始的值是 NULL 还是空字符串，返回结果都为 NULL。Impala 的这种行为是为了兼容 Hive 中分区表对 NULL 的处理。Hive 不允许空字符串作为分区键，如果查询返回了这样的值，Hive 将返回一个字符串值__HIVE_DEFAULT_PARTITION__替代 NULL 值。

示例：

```
[hadoop-cs1:21000] > create table t1 (i int) partitioned by (x int, y string);
Query: create table t1 (i int) partitioned by (x int, y string)

Returned 0 row(s) in 0.22s
[hadoop-cs1:21000] > insert into t1 partition(x=NULL, y=NULL) select id from test;
Query: insert into t1 partition(x=NULL, y=NULL) select id from test
Inserted 1 rows in 2.52s

[hadoop-cs1:21000] > insert into t1 partition(x, y=NULL) select id,id from test;
Query: insert into t1 partition(x, y=NULL) select id,id from test
Inserted 1 rows in 0.91s

-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/t1
Found 3 items
drwxrwxrwt   - impala hive          0 2014-11-20 14:06
/user/hive/warehouse/t1/.impala_insert_staging
drwxr-xr-x   - impala hive          0 2014-11-20 14:06 /user/hive/warehouse/t1/x=1
drwxr-xr-x   - impala hive          0 2014-11-20 14:03
/user/hive/warehouse/t1/x=__HIVE_DEFAULT_PARTITION__
```

（4）不提供 NOT NULL 语法定义一个非 NULL 列。

（5）不提供 DEFAULT 语法指定非空默认值。

（6）在 INSERT 插入数据时，没有指定的列将默认插入 NULL 值。

（7）在 Impala 1.2.1 或者更高版本中使用 ORDER BY … ASC 对列进行升序排序时，所有的 NULL 值将被排在最后，同样的使用 ORDER BY … DESC 对列进行降序排序时，所有的 NULL 值将被排在结果集的开头。事实上，Impala 认为 NULL 是比所有其他值更大的值。之前版本的 Impala 即使使用 ORDER BY … DESC 降序排序也始终把 NULL 值放在结果集的最后。自 Impala 1.2.1 开始，NULL 新的处理方式将更好地与其他大部分数据库保持兼容。在 Impala 1.2.1 或者更高版本中，我们也可以在 ORDER BY 语句后面使用 NULLS FIRST 或者 NULLS LAST 子句改变 NULL 默认的排序行为。

（8）除了使用 ORDER BY 排序的情况，当 NULL 值和其他任何值比较时都将返回 NULL，因为这样的比较本来就没有什么实际意义。例如： 10 > NULL 将返回 NULL，10<NULL 会返回 NULL，5 BETWEEN 1 AND NULL 也会返回 NULL。

```
[hadoop-cs1:21000] > select * from test;
Query: select * from test
+----+--------+
| id | ismale |
```

```
+----+--------+
| 1  | true   |
+----+--------+
Returned 1 row(s) in 0.33s
[hadoop-cs1:21000] > select * from test where id between 1 and null;
Query: select * from test where id between 1 and null

Returned 0 row(s) in 0.30s
[hadoop-cs1:21000] >
```

4.4 SQL 操作符

SQL 操作符确切的说是一组比较函数，被广泛使用于 SELECT 语句的 WHERE 子句中。

4.4.1 BETWEEN 操作符

在 WHERE 子句中，要比较的表达式总是有一个上限和下限。比较总是在表达式大于等于下限并且小于等于上限时匹配成功。如果上限值与下限值对换，也就是下限值大于上限值得情况下，比较将匹配不到任何值。

语法： 表达式 BETWEEN 下限值 AND 上限值

数据类型：使用 BETWEEN AND 最典型的情况是使用数字类型。实际上，我们可以使用任何数据类型，即使使用 BOOLEAN 都是可以的（BETWEEN false AND true 将匹配所有的布尔值）。如果上限值和下限值数据类型不兼容，需要使用 CAST() 进行强制转换。如果上限值和下限值只是部分不同，我们可以使用字符串或日期函数只提取相应的部分进行比较。当然，最理想的情况，还是可以把值转换成数字类型进行比较。

注意事项：当使用短字符串比较时，我们应该特别注意。一个比上限值对应的字符串更长的字符串不会被包含在 BETWEEN AND 中，因为 Impala 认为更长的字符串总是比短字符串对应的值更大。例如：BETWEEN 'A' and 'M' 将不会匹配到字符串值'Midway'。如果我们在处理字符串时想得到预期的结果，可以使用 UPPPER()、LOWER()、SUBSTR()、TRIM()等函数协助处理。

示例：

```
[hadoop-cs1:21000] > create table test(a string);
Query: create table test(a string)

Returned 0 row(s) in 0.10s
[hadoop-cs1:21000] > insert into test values("a"),("ab"),("abc"),("abcd");
```

```
Query: insert into test values("a"),("ab"),("abc"),("abcd")
Inserted 4 rows in 1.71s
[hadoop-cs1:21000] > select * from test;
Query: select * from test
+------+
| a    |
+------+
| a    |
| ab   |
| abc  |
| abcd |
+------+
Returned 4 row(s) in 0.38s
[hadoop-cs1:21000] > select * from test where a between "ab" and "abc";
Query: select * from test where a between "ab" and "abc"
+-----+
| a   |
+-----+
| ab  |
| abc |
+-----+
Returned 2 row(s) in 0.28s
```

4.4.2　比较操作符

Impala 支持为了检查列数据的等价性及大小顺序的比较操作符：

- =、 !=、 <>：所有数据类型都可以使用。
- <、<=、>、>=：所有数据类型都可以使用。对于 BOOLEAN 值，TRUE 总是大于 FALSE。
- IN、BETWEEN：使用 IN 或者 BETWEEN 操作符，可以实现对于等值、大于、小于比较等上述符号表示更简单的表达，使用单个的操作符就足够了。
- NULL：对于 NULL 值的比较也需要特别注意，NULL 值无论跟谁比较结果都是 NULL，如果要判定一个值是否为 NULL 值，需要使用 IS NULL 或者 IS NOT NULL 操作符。

```
[hadoop-cs1:21000] > select * from test where true>false;
Query: select * from test where true>false
+------+
| a    |
+------+
```

```
| a    |
| ab   |
| abc  |
| abcd |
+-------+
Returned 4 row(s) in 0.27s
```

4.4.3 IN 操作符

IN 操作符将传入的值与一组值进行比较，如果能够与其中的任意一个值匹配，就会返回TRUE。传入的值必须与给定的这组值具有兼容的数据类型。

任何使用 IN 操作符的表达式都可以被重写成使用 OR 连接的一组等值比较，但是使用 IN 操作符更清晰，更简洁，也更容易被 Impala 优化器识别。在对分区表的查询中，我们经常使用 IN 将分区列与特定值比较进行数据的过滤。

示例：

```
[hadoop-cs1:21000] > select * from test where a in("a","abc");
Query: select * from test where a in("a","abc")
+-----+
| a   |
+-----+
| a   |
| abc |
+-----+
Returned 2 row(s) in 0.28s
```

4.4.4 IS NULL 操作符

IS NULL 操作符与 IS NOT NULL 操作符正好相反，用来判断给定的值是否为 NULL。因为使用 NULL 值与任何像=、!=的比较操作符都会返回 NULL，所以当我们需要判定一个值是否为 NULL 时，需要使用这个操作符。

使用注意点：

在很多情况下，NULL 值的出现意味着数据获取或转换的过程不正确或者处理过程不完整。我们可以通过判定某列是否为 NULL，来确定是否要对这些数据进行后续的某些操作。

对于那些包含稀疏数据的宽表，非 NULL 值可能占极少数，而 NULL 值占绝大多数。如果我们要找到那些包含数据行而不关注数据本身是什么，就可以使用 IS NOT NULL 操作符。

在数据库中，我们经常使用 0、-1、’N/A’、空字符串等业务上不存在的值来表示 NULL，而

我们可以通过使用 NULL 值以及 IS NULL、IS NOT NULL 操作符来有效地避免这种自定义的设计。NULL 值能够让我们更好地区分 0、FALSE 和空，确定未知值之间的不同。

示例：

```
[hadoop-cs1:21000] > select * from test where a is not null;
Query: select * from test where a is not null
+-------+
| a     |
+-------+
| a     |
| ab    |
| abc   |
| abcd  |
+-------+
Returned 4 row(s) in 0.29s
```

4.4.5　LIKE 操作符

STRING 数据的比较操作符，其中通配符：_表示匹配单个字符，% 表示匹配多个字符。参数表达式必须与整个字符串相匹配。一般情况下，使用%匹配字符串的结尾部分效率更高。

示例：

```
[hadoop-cs1:21000] > select * from test where a like 'ab_';
Query: select * from test where a like 'ab_'
+-----+
| a   |
+-----+
| abc |
+-----+
Returned 1 row(s) in 0.32s
[hadoop-cs1:21000] > select * from test where a like 'ab%';
Query: select * from test where a like 'ab%'
+-------+
| a     |
+-------+
| ab    |
| abc   |
| abcd  |
+-------+
```

```
Returned 3 row(s) in 0.29s
```

4.4.6 REGEXP 操作符

REGEXP 操作符用来检查一个值是否与一个正则表达式相匹配。在 POSIX 正则表达式语法中，^用来匹配字符串的开头，$用来匹配字符串的结尾，.是用来匹配单个字符的通配符，*是用来匹配多个字符串的通配符，+用来匹配前面正则表达式的一个或多个实例，?用来匹配前面正则表达式的零个或一个实例。

正则表达式必须整值匹配，不能只匹配其中的某一部分。如果只需要匹配值中间的某一部分，我们需要使用.或者*来表示这一部分前面和后面的字符。因此，虽然我们可能已经在表示式中使用了^ 或$ 来表示开头或者结尾，但很多时候都是多余的。

RLIKE 操作符是 REGEXP 的同义词。

当我们需要使用()匹配多个正则表达式模式的时候，需要使用 | 作为多个操作符的分隔符。()中的模式不允许反向引用。如果要抽取()中得一部分，我们要使用内嵌函数 regexp_extract()。

示例：

```
[hadoop-cs1:21000] > select * from test where a regexp 'a.*';
Query: select * from test where a regexp 'a.*'
+------+
| a    |
+------+
| a    |
| ab   |
| abc  |
| abcd |
+------+
Returned 4 row(s) in 0.27s
[hadoop-cs1:21000] > select * from test where a regexp '^a.*d$';
Query: select * from test where a regexp '^a.*d$'
+------+
| a    |
+------+
| abcd |
+------+
Returned 1 row(s) in 0.28s
```

另外，也可以使用 reglike 操作符，它是 regexp 的别名。

4.5 模式对象和对象名称

在 Impala 中，我们使用的模式对象与传统数据库用户使用的很相似，包括数据库，表，视图和函数。

在表的概念中，分区也是一种类型的对象。事实上，分区是 Impala 非常重要的一个主题。

不像其他的数据库系统，Impala 没有表空间的概念。默认情况下，一个数据库，表或者分区对应的所有的数据文件都会存储在 HDFS 的某层文件夹中。我们也可以为某个表或者分区指定存储在 HDFS 的什么位置上。这些对象的原始数据通常是一组数据文件，我们可以通过简单地将这组数据文件移动到 HDFS 上指定的位置提高数据加载的灵活性。

所有关于模式对象的信息都存储在元数据库中。Impala 和 Hive 的元数据保持兼容，在 Hive 中创建的表可以在 Impala 中使用，反之亦然。当 Impala 使用 CREATE、ALTER、INSERT、LOAD DATA 等改变模式对象时，所有的元数据的变化将会通过 catalog 服务把变化广播到集群的每一个节点上。当我们使用 Hive 或者直接操作 HDFS 文件改变元数据定义时，在 Impala 中需要通过 REFRESH 或 INVALIDATE METADATA 来更新自己的元数据。

4.5.1　别名

当我们在查询中使用表、列或者列表达式时，可以给它们指定别名。在同一个语句中多次引用表、列时，我们可以不使用它原始的名称，而是使用别名。通常情况下，我们使用的别名比原始名称更短，更直观。别名放在查询结果的头部，使查询结果更通俗易懂。

我们可以在 SELECT 列表或者 FROM 列表中的表、列、列表达式的后面紧跟上“AS 别名”的语法来设置一个别名。实际上，在原始的名后面紧跟上别名，也是可以的，AS 关键字是可选的。

示例：

```
[hadoop-cs1:21000] > select a as name from test;
Query: select a as name from test
+------+
| name |
+------+
| a    |
| ab   |
| abc  |
| abcd |
+------+
Returned 4 row(s) in 0.29s
```

```
[hadoop-cs1:21000] > select name from (select a as name from test) student;
Query: select name from (select a as name from test) student
+-------+
| name |
+-------+
| a    |
| ab   |
| abc  |
| abcd |
+-------+
Returned 4 row(s) in 0.29s
```

在大小写不敏感的问题上，别名和标示符遵循相同的规则。别名可以比标示符更长（最长可以是 Java 字符串的最大长度），而且别名在使用反引号引用时，可以包含空格、破折号等。

要让一个表具有不同的名称，除了别名，还可以通过视图实现。

4.5.2 标示符

标示符指的是我们可以在 SQL 中使用的数据库、表或者列的名字。标示符的规则定义了我们可以为创建的对象指定什么样的名称，对包含特殊字符的相关名称如何表示及名称大小写是否敏感等。

- 标示符最小长度为 1 个字符。
- 标示符的最大长度由于元数据库的限制，为 128 个字符。
- 标示符必须以字母字符开始，剩下的字符可以为字母、数字、下划线的任何组合。对一个合法的标示符使用反引号引用，对它的使用没有影响。
- 标示符只能包含 ASCII 字符。
- 为了使用一个与 Impala 预留关键字匹配的标示符，需要使用反引号引用。

Impala 标示符大小写不敏感，也就是说表 T1 和表 t1 对应的是同一张表。在底层，Impala 总是将表或者列名转换成小写字符，所以查询输出的数据表头都是小写的。

4.5.3 数据库

在 Impala 中，数据库是一组表的逻辑容器。每个数据库定义一个不同的命名空间。在一个数据库内部，我们可以使用非限定名称引用表。不同的数据库可以包含具有相同名称的表。

创建数据库是一个轻量级的操作，不需要为数据库指定数据库配置属性。因此，没有 ALTER DATABASE 命令。

一般情况下，为了避免表的命名冲突，为了让表的关联关系更清晰，我们会根据不同的业务

或者不同的项目建立单独的数据库。

每个数据库物理上存储在 HDFS 的一个目录中。

当我们最开始连接到 Impala 上时，默认情况下会连接到一个名称为 default 的数据库。在 default 库中创建的表在 HDFS 上比其他用户创建的表具有更高的级别。

```
-bash-4.1$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > show databases;
Query: show databases
+---------------------+
| name                |
+---------------------+
| _impala_builtins    |
| default             |
| external_partitions |
+---------------------+
Returned 3 row(s) in 0.01s
 [hadoop-cs1:21000] > select current_database();
Query: select current_database()
+--------------------+
| current_database() |
+--------------------+
| default            |
+--------------------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] >
```

4.5.4 表

表是数据的主要容器。与传统的数据库系统类似，表也按行、列分布，也具备传统数据库的中表的特性。比如在高端的数据仓库系统中经常会使用表的分区特性。

逻辑上，每个表基于它的列、分区和其他的属性定义有一个结构。

物理上，每个表与 HDFS 的一个目录相关联。表的数据包括在这个目录下的所有数据文件。

- 内部表：这种表由 Impala 管理，使用 Impala 工作区指定的目录。
- 外部表：这种表使用任意的 HDFS 目录作为数据存储，目录中的数据文件可以被不同的 Hadoop 组件共享。
- 大表：通常表现为分区表，数据文件被分离放在不同的 HDFS 子目录中。

1. 内部表

通过 CREATE TABLE 语句创建的默认的表类型为内部表。

Impala 在 HDFS 上创建一个目录存放数据文件。我们可以通过 impala-shell 使用 INSERT 语句或者通过 Hive 使用 LOAD DATA 语句向其中加载数据。当我们执行了 DROP TABLE 语句后，Impala 将自动的删除所有的数据文件。

2. 外部表

语法 CREATE EXTERNAL TABLE 可以创建一个 Impala 表，这个表的数据来自正常的 Impala 数据目录之外的已经存在的数据文件。当我们在 HDFS 的某个位置已经有了数据文件时，使用外部表的方式将节省了把原有数据导入新表的开销。

我们可以使用 Impala 查询表中的数据。如果我们使用 HDFS 操作添加或者替换了数据，需要在 impala-shell 中使用 REFRESH 命令让 Impala 同步数据文件、块位置等元数据信息。当我们执行 DROP TABLE 语句时，Impala 将删除表与相关数据文件的关系，但是不会从物理上删除数据文件。我们仍然可以在 Hadoop 其他组件中使用这些文件。

4.5.5 视图

视图是可以作为别名查询的轻量级的逻辑结构。我们可以在查询中使用表明的位置指定一个视图名称。视图的作用如下：

（1）可以是实现更细粒度的访问控制，用户只能查询到允许他看到的列，而不能查询其他列。

（2）使用简洁的语法完成复杂查询。

```
    [hadoop-cs1:21000] > select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit
5;
    Query: select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit 5
    +----+----+---------+
    | c1 | c2 | avg(c3) |
    +----+----+---------+
    | 2  | 3  | 1.5     |
    | 3  | 4  | 1.5     |
```

```
+----+----+---------+
Returned 2 row(s) in 0.55s

[hadoop-cs1:21000] > create view v1 as select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit 5;
Query: create view v1 as select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit 5

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > select * from v1;
Query: select * from v1
+----+----+-----+
| c1 | c2 | _c2 |
+----+----+-----+
| 2  | 3  | 1.5 |
| 3  | 4  | 1.5 |
+----+----+-----+
Returned 2 row(s) in 1.81s
```

（3）通过向原始查询添加新的语句、SELECT 列表表达式、函数调用等创建一个新的更精确的查询。

```
[hadoop-cs1:21000] > create view v1 as select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit 5;
Query: create view v1 as select c1, c2, avg(c3) from t1 group by c1,c2 order by c1 limit 5

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > create view v2 as select * from v1 order by _c2 desc limit 5;
Query: create view v2 as select * from v1 order by _c2 desc limit 5

Returned 0 row(s) in 0.22s
```

这项技术让我们对同一个查询进行或多或少的变化，以实现不同功能的需求。

（4）为表、列、join 的结果集等指定更直观的别名：

```
[hadoop-cs1:21000] > create view v1 as select c1 as ChineseRecord, c2 as EnglishRecord, avg(c3) as AvgRecord from t1 group by c1,c2;
Query: create view v1 as select c1 as ChineseRecord, c2 as EnglishRecord, avg(c3)
```

```
as AvgRecord from t1 group by c1,c2

   Returned 0 row(s) in 0.20s
```

（5）将表与其他不同文件格式，不同分区模式的表交换，而无须考虑数据复制及转换的停机时间。

```
   [hadoop-cs1:21000] >
   [hadoop-cs1:21000] > create table slow (x int, s string) stored as textfile;
   Query: create table slow (x int, s string) stored as textfile

   Returned 0 row(s) in 0.11s
   [hadoop-cs1:21000] > create view report as select s from slow where x between 20
and 30;
   Query: create view report as select s from slow where x between 20 and 30

   Returned 0 row(s) in 1.07s
   [hadoop-cs1:21000] > insert into slow values(21,'abcd'),(25,'jcq0');
   Query: insert into slow values(21,'abcd'),(25,'jcq0')
   Inserted 2 rows in 0.40s
   [hadoop-cs1:21000] > select * from report;
   Query: select * from report
   +------+
   | s    |
   +------+
   | abcd |
   | jcq0 |
   +------+
   Returned 2 row(s) in 1.17s
   [hadoop-cs1:21000] > create table fast (s string) partitioned by (x int) stored
as parquet;
   Query: create table fast (s string) partitioned by (x int) stored as parquet

   Returned 0 row(s) in 0.20s
   [hadoop-cs1:21000] > insert into fast partition(x=22) values('defg');
   Query: insert into fast partition(x=22) values('defg')
   Inserted 1 rows in 0.71s
   [hadoop-cs1:21000] > insert into fast partition(x=28) values('cjq0');
   Query: insert into fast partition(x=28) values('cjq0')
   Inserted 1 rows in 0.71s
```

```
[hadoop-cs1:21000] > alter view report as select s from fast where x between 20
and 30;
Query: alter view report as select s from fast where x between 20 and 30
[hadoop-cs1:21000] > select * from report;
Query: select * from report
+------+
| s    |
+------+
| defg |
| cjq0 |
+------+
Returned 2 row(s) in 0.28s
[hadoop-cs1:21000] >
```

为了显示视图的定义，我们使用 DESCRIBE FORMATTED 语句，它可以显示原始的 CREATE VIEW 语句。

```
[hadoop-cs1:21000] > desc v1;
Query: describe v1
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| c1   | int    |         |
| c2   | int    |         |
| _c2  | double |         |
+------+--------+---------+
Returned 3 row(s) in 0.01s
[hadoop-cs1:21000] > desc formatted v1;
Query: describe formatted v1
+------------------------------+--------------------------------------------
-------------------------+----------------------+
| name                                                                | type
| comment              |
+------------------------------+--------------------------------------------
-------------------------+----------------------+
| # col_name                                                   | data_type
| comment              |
|                                                                     | NULL
| NULL                 |
| c1                                                                   | int
```

```
| None                |
   |   c2                                                                      |   int
| None                |
   |   _c2                                                                 |   double
| None                |
   |                                                                        |   NULL
| NULL                |
   |        #        Detailed        Table        Information        |        NULL
| NULL                |
   |   Database:                                                        |   default
| NULL                |
   |   Owner:                                                            |   impala
| NULL                |
   |  CreateTime:                              |  Thu  Nov  20  15:07:30  CST  2014
| NULL                |
   |    LastAccessTime:                                                 |   UNKNOWN
| NULL                |
   |    Protect    Mode:                                                  |    None
| NULL                |
   |    Retention:                                                           |    0
| NULL                |
   |   Table    Type:                                                |   VIRTUAL_VIEW
| NULL                |
   |     Table     Parameters:                                           |     NULL
| NULL                |
   |                                                   |   transient_lastDdlTime
| 1416467250           |
   |                                                                        |   NULL
| NULL                |
   |     #     Storage     Information                                   |     NULL
| NULL                |
   |     SerDe     Library:                                                      |
| NULL                |
   |     InputFormat:                                                             |
| NULL                |
   |     OutputFormat:                                                            |
| NULL                |
   |    Compressed:                                                       |     No
| NULL                |
```

```
    |    Num    Buckets:                                                    |    0
| NULL                 |
    |    Bucket    Columns:                                                 |    []
| NULL                 |
    |    Sort    Columns:                                                   |    []
| NULL                 |
    |                                                                       |   NULL
| NULL                 |
    |    #    View    Information                                           |   NULL
| NULL                 |
    | View Original Text:        | SELECT c1, c2, avg(c3) FROM t1 GROUP BY c1, c2 ORDER
BY c1 ASC LIMIT 5 | NULL                 |
    | View Expanded Text:        | SELECT c1, c2, avg(c3) FROM t1 GROUP BY c1, c2 ORDER
BY c1 ASC LIMIT 5 | NULL                 |
    +-------------------------------+------------------------------------------------
--------------------------+----------------------+
    Returned 29 row(s) in 0.16s
```

视图有如下限制：

- 不能向 Impala 视图进行插入操作。在某些数据库中，允许通过视图向基表插入数据，但 Impala 不支持该操作。在 INSERT 语句右侧的 SELECT 子句中可以使用视图名称。
- 如果视图对应的基表为分区表，分区修剪的优化操作将依赖于原始的查询语句。如果是针对视图的 WHERE 条件列中包含了分区键，Impala 不会进行分区修剪。

4.5.6　函数

函数使我们可以让 Impala 数据进行数学、字符串及其他计算或者转换操作。我们可以在 SELECT 列表和 WHERE 子句中使用函数进行过滤操作或者格式化查询结果，这样就避免了对查询结果在应用侧的进一步处理。

4.6　SQL 语句

Impal 支持标准 SQL，同时它在大数据的数据加载和数据仓库方面进行了一些扩展。与标准 SQL 语言一样，Impal 的 SQL 语句也分为数据定义语言和数据操纵语言。

DDL 是“Data Definition Language”的缩写，指的是数据定义语言，通过它可以改变数据库模式的结构定义，DDL 是 SQL 语句的一个子集。通过 DDL 语句，我们可以创建、删除、修改数

据库、表、视图等对象定义。大多数的 DDL 语句以关键字 CREATE、DROP、ALTER 开头。

在 Impala 执行了 DDL 语句之后，和对应的表、列、视图、分区等的元数据信息将会自动的被同步到集群的所有节点上。

如果元数据更新的时间比较长，那么我们可以启用 SYNC_DDL 查询选项。启用这个选项后，DDL 语句会等待所有节点都收到元数据变化后才会执行。比如，如果我们通过 ROUND-ROBIN 算法调度查询任务，每次将任务分发到一个不同的 Impala 节点上时可以启用这个查询选项。

INSERT 语句在传统数据库中一直被认为是 DML 语句，但是在 Impala 中 INSERT 操作也会引起元数据发生变化，这种变化时也会像所有的Impala节点广播。SYNC_DDL 查询选项对INSERT 引起的元数据变化同样有效。

由于 SYNC_DDL 查询选项会让每个 DDL 操作消耗更多的时间，所以我们建议只在最后一个 DDL 操作之前使用。如果我们要运行一组 DDL 操作的脚本，从性能上考虑，我们只需要在最后的 CREATE、DROP、ALTER 或者 INSERT 上启用 SYNC_DDL 选项即可。只有所有的 Impala 节点接收到这些元数据的变化，脚本才会执行结束。

Impala 和 Hive 对 DDL、DML 和其他语句的分类并不完全相同。Impala 组织这些语句的方式跟传统的关系型数据库和数据仓库更为类似。像 COMPUTE STATS 这样修改元数据库的语句，被称为 DDL 语句。像 SHOW、DESCRIBE 这样的只查询元数据库的语句，被划分为单独的一类。

DML 是“Data Manipulation Language”的缩写，指的是数据操纵语言，是 SQL 语句的一个子集。通常使用 DML 语句修改表中的数据。由于 Impala 解决的是查询性能的问题，而不是数据操纵的问题，而 HDFS 先天只支持追加操作，所以目前 Impala 只支持少部分 DML 操作。

在 Impala 中，使用单个语句进行大批量数据的插入能够更有效的使用 HDFS 块。如果我们使用 INSERT…VALUES 一次插入一行或者几行数据，可能将表存储在 HBase 中更有优势。

为了模拟其他数据库中的 UPDATE 或者 DELETE 操作，我们需要使用 INSERT 或者 CREATE TABLE AS SELECT 语句将原有的数据过滤，转换后复制到一个新的表中。

虽然 Impala 不支持 UPDATE 语句，但是我们可以通过将表存储在 HBase 中变相的实现 UPDATE 的功能。当我们向 HBase 中插入一行已经存在该键值的记录时，原有的记录将会被隐藏，效果和 UPDATE 是一样的。

4.6.1 ALTER TABLE

ALTER TABLE 语句可以用来修改一个已存在的表的结构和属性。在 Impala 中，执行这个语句会更新元数据库中的元数据表，但是不会修改真正的数据文件。因为我们需要手动的执行对数据文件的操作，比如讲数据文件移动到不同的 HDFS 目录，通过重写数据文件增加字段，转换不同的文件格式等。

语句类型：DDL

重命名表：

```
ALTER TABLE old_name RENAME TO new_name;
```

在底层，这个操作只是修改了包含数据文件的 HDFS 目录名称，修改之后原始的目录名称就

不存在了。通过在表名前面指定数据库名，我们可以把一张表从一个数据库移动到另一个数据库。

```
[hadoop-cs1:21000] > create database d1;
Query: create database d1

Returned 0 row(s) in 0.28s
[hadoop-cs1:21000] > create database d2;
Query: create database d2

Returned 0 row(s) in 0.08s
[hadoop-cs1:21000] > create database d3;
Query: create database d3

Returned 0 row(s) in 0.17s
[hadoop-cs1:21000] > use d1;
Query: use d1
[hadoop-cs1:21000] > create table mobile (x int);
Query: create table mobile (x int)

Returned 0 row(s) in 0.10s
[hadoop-cs1:21000] > use d2;
Query: use d2
[hadoop-cs1:21000] > alter table d1.mobile rename to mobile;
Query: alter table d1.mobile rename to mobile
[hadoop-cs1:21000] > use d1;
Query: use d1
[hadoop-cs1:21000] > alter table d2.mobile rename to d3.mobile;
Query: alter table d2.mobile rename to d3.mobile
[hadoop-cs1:21000] > use d3;
Query: use d3
[hadoop-cs1:21000] > show tables;
Query: show tables
+--------+
| name   |
+--------+
| mobile |
+--------+
Returned 1 row(s) in 0.02s
[hadoop-cs1:21000] >
```

如果需要改变 Impala 的表对应的数据文件的物理位置，需使用以下语句：

```
ALTER TABLE table_name SET LOCATION 'hdfs_path_of_directory';
```

我们在'hdfs_path_of_directory'中指定的 HDFS 路径将会自动被创建，Impala 不会在该路径下创建子目录。同时，Impala 不会改变指定的 HDFS 目录中已经存在的数据文件，也不会向这个目录中移动任何文件。

如果需要改变表属性或者存储属性，需使用以下语句：

```
ALTER TABLE table_name SET TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...);
ALTER TABLE table_name SET SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...);
```

TBLPROPERTIES 主要是指定表级别的特定属性。

SERDEPROPERTIES 主要是通过 Hive 来实现对表的读写方式等元数据的定义，在 Impala 中定义这个属性的情况并不多见。通过这个语句可以指定参数'serialization.format'或'field.delim'来改变已存在的表或分区的分隔符。

```
[hadoop-cs1:21000] > use default;
Query: use default
[hadoop-cs1:21000] > create table change_to_csv (s1 string, s2 string) row format
delimited fields terminated
                  > by '|';
Query: create table change_to_csv (s1 string, s2 string) row format delimited fields
terminated by '|'

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > alter table change_to_csv set SERDEPROPERTIES
('serialization.format'=',',
                  > 'field.delim'=',');
Query: alter table change_to_csv set SERDEPROPERTIES ('serialization.format'=',',
'field.delim'=',')
[hadoop-cs1:21000] > insert overwrite change_to_csv values ('stop','go'),
('yes','no');
Query: insert overwrite change_to_csv values ('stop','go'), ('yes','no')
Inserted 2 rows in 0.41s

-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/change_to_csv
Found 2 items
drwxrwxrwt   -   impala   hive          0   2014-11-20   17:19
/user/hive/warehouse/change_to_csv/.impala_insert_staging
-rw-r--r--   3   impala   hive         15   2014-11-20   17:19
```

```
/user/hive/warehouse/change_to_csv/364952a2d7d32c19-e143eb9e15bc0ebf_1603390261_d
ata.0
   -bash-4.1$                    hdfs                    dfs                   -cat
/user/hive/warehouse/change_to_csv/364952a2d7d32c19-e143eb9e15bc0ebf_1603390261_d
ata.0
   stop,go
   yes,no
```

通过 DESCRIBE FORMATTED 语句可以查看表的当前各属性值。

```
   [hadoop-cs1:21000] > desc formatted change_to_csv;
   Query: describe formatted change_to_csv
   +------------------------------+-----------------------------------------------
--------------+----------------------+
   | name                        | type                                          |
comment               |
   +------------------------------+-----------------------------------------------
--------------+----------------------+
   |   #   col_name                                                 |   data_type
|  comment            |
   |                           | NULL                                          |
NULL                 |
   | s1                        | string                                        |
None                 |
   | s2                        | string                                        |
None                 |
   |                           | NULL                                          |
NULL                 |
   |        #        Detailed        Table        Information        |        NULL
| NULL                |
   |    Database:                                                     |    default
| NULL                |
   |    Owner:                                                         |    impala
| NULL                |
   |   CreateTime:                              |  Thu  Nov  20  17:18:42  CST  2014
| NULL                |
   |     LastAccessTime:                                              |    UNKNOWN
| NULL                |
   |     Protect    Mode:                                               |      None
| NULL                |
```

```
    |   Retention:                                                          |   0
| NULL                 |
    |   Location:                                                                |
hdfs://hadoop-cm1:8020/user/hive/warehouse/change_to_csv   | NULL              |
    |  Table   Type:                                              |  MANAGED_TABLE
| NULL                 |
    |   Table    Parameters:                                              |    NULL
| NULL                 |
    |                                                         |  transient_lastDdlTime
| 1416475134             |
    |                          | NULL                                            |
NULL                 |
    |    #    Storage    Information                                        |     NULL
| NULL                 |
    |   SerDe    Library:                                                          |
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe       | NULL               |
    | InputFormat:                       |  org.apache.hadoop.mapred.TextInputFormat
| NULL                 |
    |   OutputFormat:                                                              |
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL               |
    |   Compressed:                                                         |    No
| NULL                 |
    |   Num    Buckets:                                                     |    0
| NULL                 |
    |   Bucket    Columns:                                                  |    []
| NULL                 |
    |   Sort    Columns:                                                    |    []
| NULL                 |
    |    Storage    Desc    Params:                                         |    NULL
| NULL                 |
    |                                                               |   field.delim
| ,                    |
    |                                                          |  serialization.format
| ,                    |
    +-------------------------------+----------------------------------------------
--------------+---------------------+
```

我可以通过以下语句实现对列的属性的修改：

```
ALTER TABLE table_name ADD COLUMNS (column_defs);
```

```
ALTER TABLE table_name REPLACE COLUMNS (column_defs);
ALTER TABLE table_name CHANGE column_name new_name new_spec;
ALTER TABLE table_name DROP column_name;
```

语句中的 column_defs 与 CREATE TABLE 语句中的列定义相同，包括：列名称，列数据类型，也可以包含列的注释。我们可以一次为一张表添加多个列。无论添加单列或者多列，都需要用括弧将列描述信息括起来。当我们替换一列的时候，该列原有的定义信息将会彻底删除。如果新增的数据文件中对应的列发生了变化，或者列数据类型发生了变化，则需要使用上述语句对列的属性进行修改。如果修改过得列与原有列的数据类型不兼容，则必须通过 INSERTOVERWRITE 或 LOAD DATA OVERWRITE 将原有列数据类型对应的数据替换，否则将无法对该列正常查询。

我们可以通过 CHANGE 语句重命名一个列，或者修改一个列的数据类型。比如将列由 STRING 修改为 TIMESTAMP，或者从 INT 修改为 BIGINT。我们每次只能删除一列，如果要想删除多列，需要多次运行 ALTER TABLE table_name DROP colume_name 语句。如果我们要定义新的列，可以使用 ALTER TABLE ... REPLACE COLUMNS 语句。

为了改变 Impala 表底层对应的数据文件的文件格式，我们可以使用如下命令：

```
ALTER TABLE table_name SET FILEFORMAT { PARQUET | PARQUETFILE | TEXTFILE | RCFILE
|SEQUENCEFILE }
```

需要注意的是，这个语句只会改变 Impala 表的元数据定义，表对应的数据文件的文件格式的转换需要使用 Impala 之外的技术实现，但是通过 Impala 的 INSERT 命令插入的数据将自动会存储为新的文件格式。对于 TEXTFILE，我们无法为其指定分隔符，它必须使用逗号分隔。

只有针对分区表（使用 PARTITIONED BY 语句创建的表），我们才能进行添加或者删除分区的操作。表的分区在 HDFS 上表现为以分区键对应的具体值命名的目录。在通过移动或者拷贝数据文件到对应的目录的方式导入数据后，ALTER TABLE... ADD PARTITION 语句主要用来实现表分区与数据文件的关联。与之相反，ALTER TABLE... DROP PARTITION 语句用来删除分区和数据文件的这种关联关系。比如，如果我们只需要三个月的数据，那么在每个月月初，我们都需要删除三个月之前那个月的数据。删除分区操作可以减少对应表的元数据的数据量，加快分区表的查询操作，尤其对于连接查询可以降低执行计划计算的复杂度。添加和删除分区的示例如下：

```
[hadoop-cs1:21000] > create table part_t (x int) partitioned by (month string);
Query: create table part_t (x int) partitioned by (month string)

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > alter table part_t add partition (month='January');
Query: alter table part_t add partition (month='January')
[hadoop-cs1:21000] > refresh part_t;
Query: refresh part_t

Returned 0 row(s) in 0.25s
```

```
[hadoop-cs1:21000] > alter table part_t add partition (month='February');
Query: alter table part_t add partition (month='February')
[hadoop-cs1:21000] > alter table part_t drop partition (month='January');
Query: alter table part_t drop partition (month='January')
[hadoop-cs1:21000] > alter table part_t partition (month='February') set
fileformat parquet;
Query: alter table part_t partition (month='February') set fileformat parquet
```

分区键也可以引用列的任意常量表达式。

```
alter table time_data add partition (month=concat('Decem','ber'));
alter table sales_data add partition (zipcode = cast(9021 * 10 as string));
```

注意事项：

针对特定的分区使用 ALTER TABLE 语句时，我们必须包含表中定义的所有分区列。

对于内部表和外部表，针对 ALTER TABLE 操作的大部分运行机制都是一样的。唯一不同的是外部表不会依赖于 ALTER TABLE 语句的修改而改变或者移动数据文件所在的目录。

如果在 impala-shell 中使用了负载均衡机制让每一次查询连接到不同的 Impala 节点上，我们可以使用 SYNC_DDL 查询选项，让每个 DDL 语句的元数据变更同步到所有 Impala 节点上后才会返回。

4.6.2 ALTER VIEW

该语句可以改变和一个视图的查询关联，该视图关联的数据库，或者视图的名称。

由于视图是一个逻辑定义，所以 ALTER VIEW 语句只会改变元数据，而不会改变 HDFS 上存放真正数据的数据文件。

语法：

```
ALTER VIEW [database_name.]view_name AS select_statement
ALTER VIEW [database_name.]view_name RENAME TO [database_name.]view_name
```

语句类型：DDL

如果在 impala-shell 中使用了负载均衡机制让每一次查询连接到不同的 Impala 节点上，我们可以使用 SYNC_DDL 查询选项，让每个 DDL 语句的元数据变更同步到所有 Impala 节点上后才会返回。

示例：

```
[hadoop-cs1:21000] > create table t1 (x int, y int, s string);
Query: create table t1 (x int, y int, s string)
```

```
Returned 0 row(s) in 0.22s
[hadoop-cs1:21000] > create table t2 like t1;
Query: create table t2 like t1

Returned 0 row(s) in 1.27s
[hadoop-cs1:21000] > create view v1 as select * from t1;
Query: create view v1 as select * from t1

Returned 0 row(s) in 0.19s
[hadoop-cs1:21000] > alter view v1 as select * from t2;
Query: alter view v1 as select * from t2
[hadoop-cs1:21000] > alter view v1 as select x, upper(s) s from t2;
Query: alter view v1 as select x, upper(s) s from t2
```

如果想看视图的定义，我们需要执行 DESCRIBE FORMATTED 语句，这样就会显示出原始的视图创建语句。

```
[hadoop-cs1:21000] > desc formatted v1;
Query: describe formatted v1
+------------------------------+------------------------------+---------------
-------+
| name                         | type                         | comment              |
+------------------------------+------------------------------+---------------
-------+
| # col_name                   | data_type                    | comment              |
|                              | NULL                         | NULL                 |
| x                            | int                          | None                 |
| s                            | string                       | None                 |
|                              | NULL                         | NULL                 |
| # Detailed Table Information | NULL                         | NULL                 |
| Database:                    | default                      | NULL                 |
| Owner:                       | impala                       | NULL                 |
| CreateTime:                  | Thu Nov 20 17:43:48 CST 2014 | NULL                 |
| LastAccessTime:              | UNKNOWN                      | NULL                 |
| Protect Mode:                | None                         | NULL                 |
| Retention:                   | 0                            | NULL                 |
| Table Type:                  | VIRTUAL_VIEW                 | NULL                 |
| Table Parameters:            | NULL                         | NULL                 |
|                              | transient_lastDdlTime        | 1416476647           |
|                              | NULL                         | NULL                 |
```

```
| # Storage Information           | NULL                        | NULL           |
| SerDe Library:                  |                             | NULL           |
| InputFormat:                    |                             | NULL           |
| OutputFormat:                   |                             | NULL           |
| Compressed:                     | No                          | NULL           |
| Num Buckets:                    | 0                           | NULL           |
| Bucket Columns:                 | []                          | NULL           |
| Sort Columns:                   | []                          | NULL           |
|                                 | NULL                        | NULL           |
| # View Information              | NULL                        | NULL           |
| View Original Text:             | SELECT x, upper(s) s FROM t2 | NULL          |
| View Expanded Text:             | SELECT x, upper(s) s FROM t2 | NULL          |
+-------------------------------+------------------------------+---------------
-------+
Returned 28 row(s) in 0.02s
```

4.6.3 COMPUTE STATS

这一语句完成对表、分区、列的数据量和数据分布信息搜集。搜集的相关信息会被存储在元数据库中。Impala 在对查询进行优化时将使用到这些统计信息。如果 Impala 知道是大表还是小表，每列值的区分度大小，将能更好地判断对于一个 JOIN 或者 INSERT 操作是否进行并行化操作。

语句类型：DDL

使用注意点：

在早期，Impala 使用 Hive 运行 ANALYZE TABLE 语句生成的信息，但是 Hive 生成的统计信息不可靠，而且不易使用。Impala 的 COMPUTE STATS 语句基于底层编码，大大提高了可靠性和用户友好度。COMPUTE STATS 不需要特别的安装或配置。不像 Hive 需要使用单独的 ANALYZE TABLE 语句分别统计表和列统计信息，Impala 只需使用一个单条语句 COMPUTE STATS 语句即可实现所有统计信息搜集。

基于 HBase 的考虑：

COMPUTE STATS 对 HBase 的表也同样起作用。搜集 HBase 表的统计信息与搜集基于 HDFS 的 Impala 表的方式有些不同，但是在对 HBase 表进行 JOIN 时仍然可以将这些元数据信息用于优化查询。

基于性能的考虑：

通过 COMPUTE STATS 搜集统计的信息，Impala 对 JOIN 查询和资源高消耗的查询进行优化。

示例：

这个例子中共使用了两张表，分别为 T1，T2。这两张表通过 T1.ID 和 T2.PARENT 两列具有父子关联关系。T1 表很小，但是 T2 表有大约 10 万条记录。在进行搜集统计信息之前，统计信息包括表的物理信息，比如文件的个数，总大小，固定列长等。对于那些未知的信息，统一用-1 表示。在运行了 COMPUTE STATS 之后，通过 SHOW STATS 可以显示更多的信息。如果需要进行一个 JOIN 查询，那么我们需要对所有参与关联的表搜集统计信息，只有这样才能保证 JOIN 查询可以生成最高效的执行计划。

```
[hadoop-cs1:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| -1 | 1 | 33B | TEXT |
+-------+--------+------+--------+
Returned 1 row(s) in 0.02s
[hadoop-cs1:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+----------+--------+
| -1 | 28 | 960.00KB | TEXT |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.01s
[hadoop-cs1:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+-----------------+--------+----------+----------+
| id | INT | -1 | -1 | 4 | 4 |
| s | STRING | -1 | -1 | -1 | -1 |
+--------+--------+-----------------+--------+----------+----------+
Returned 2 row(s) in 1.71s
[hadoop-cs1:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+-----------------+--------+----------+----------+
| parent | INT | -1 | -1 | 4 | 4 |
```

```
| s | STRING | -1 | -1 | -1 | -1 |
+--------+--------+-----------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
[hadoop-cs1:21000] > compute stats t1;
Query: compute stats t1
+--------------------------------------+
| summary |
+--------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+--------------------------------------+
Returned 1 row(s) in 5.30s
[hadoop-cs1:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| 3 | 1 | 33B | TEXT |
+-------+--------+------+--------+
Returned 1 row(s) in 0.01s
[hadoop-cs1:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+-----------------+--------+----------+----------+
| id | INT | 3 | 0 | 4 | 4 |
| s | STRING | 3 | 0 | -1 | -1 |
+--------+--------+-----------------+--------+----------+----------+
Returned 2 row(s) in 0.02s
[hadoop-cs1:21000] > compute stats t2;
Query: compute stats t2
+--------------------------------------+
| summary |
+--------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+--------------------------------------+
Returned 1 row(s) in 5.70s
[hadoop-cs1:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
```

```
| #Rows | #Files | Size | Format |
+-------+--------+----------+--------+
| 98304 | 1 | 960.00KB | TEXT |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.03s
[hadoop-cs1:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+-----------------+--------+----------+----------+
| parent | INT | 3 | 0 | 4 | 4 |
| s | STRING | 6 | 0 | -1 | -1 |
+--------+--------+-----------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
```

基于文件格式的考虑：

COMPUTE STATS 语句可以对 Impala 支持的所有文件类型的表进行统计信息搜集。无论是通过 Impala 还是 Hive 创建的表，都可以使用该语句搜集。针对不同文件类型的表，限制分别如下：

- 文本文件：针对这种类型没有任何限制。
- Parquet 表：可以正常使用 COMPUTE STATS。
- Avro 表：需要表使用 SQL 方式指定列名和类型。
- RCFile 表：针对这种类型没有任何限制。
- SequenceFile 表：针对这种类型没有任何限制。

针对分区表，无论是所有分区使用相同的数据文件类型，或者某些分区已使用 ALTER TABLE 修改为其他数据文件类型，都可以使用 COMPUTE STATS 搜集统计信息。

4.6.4　CREATE DATABASE

在 Impala 中，数据库的概念在逻辑上，是在一个单独的命名空间内的相关表的逻辑结构体的集合。我们可以针对某个应用，或者业务含义类似的一组表使用单独的数据存放；在物理上，数据库对应的是 HDFS 上的一个目录树。表，分区，数据文件都存储在这个目录中。我们可以通过这个目录来判定这个数据库对空间的占用情况，也可以通过备份这个目录来备份整个数据库，或者通过 DROP DATABASE 来删除数据库。

语法：

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name[COMMENT
'database_comment'][LOCATION hdfs_path];
```

使用注意点：

一个数据库物理上对应与一个 HDFS 目录，在 Impala 数据主目录中，有一个扩展名为.db 的文件。如果和数据库关联的 HDFS 目录不存在，Impala 将自动创建这个目录。

在创建了数据库之后，可以通过 impala-shell 访问到它。如果想让自己创建的数据库称为当前数据库，需要在命令行中使用 USE 语句。在当前数据库下访问数据库中的表不需要任何前缀。如果通过 impala-shell 连接进来之后，默认使用的数据库名称为 default。

在创建完一个数据库之后，在执行该操作的同一个节点的不同 impala-shell 中可以直接访问到该数据库。但是位于其他节点的 impala-shell 会话在访问该数据库之前需要执行 INVALIDATE METADATA 操作。如果不想每次都执行 INVALIDATE METADATA 语句，那么可以在创建数据库的 DDL 语句上启用 SYNC_DDL 查询选项。

示例：

```
[hadoop-cs1:21000] > create database first1;
Query: create database first1

Returned 0 row(s) in 0.19s
[hadoop-cs1:21000] > use first1;
Query: use first1
[hadoop-cs1:21000] > create table t1 (x int);
Query: create table t1 (x int)

Returned 0 row(s) in 0.21s
[hadoop-cs1:21000] > drop database first1;
Query: drop database first1
ERROR: AnalysisException: Cannot drop current default database: first1
[hadoop-cs1:21000] > use default;
Query: use default
[hadoop-cs1:21000] > drop database first1;
Query: drop database first1
ERROR: InternalException: Database first1 is not empty
[hadoop-cs1:21000] > drop table first1.t1;
Query: drop table first1.t1
[hadoop-cs1:21000] > drop database first1;
Query: drop database first1
[hadoop-cs1:21000] >
```

4.6.5 CREATE FUNCTION

该语句用来创建自定义的函数（UDF），这样，我们就可以在 SELECT 或者 INSERT 中执行

自己实现的逻辑。

语法：

对于针对每行都要调用一次的自定义函数，称为标量自定义函数：

```
CREATE FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[, arg_type...])
RETURNS return_type
LOCATION 'hdfs_path'
SYMBOL='symbol_or_class'
```

对于使用多个函数实现，需要跨多行计算的自定义函数，称为自定义聚集函数：

```
CREATE [AGGREGATE] FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[,
arg_type...])
RETURNS return_type
LOCATION 'hdfs_path'
[INIT_FN='function]
UPDATE_FN='function
MERGE_FN='function
[PREPARE_FN='function]
[CLOSEFN='function]
[FINALIZE_FN='function]
```

语句类型：DDL

最简单的自定义函数就是每次对单行调用返回一行的标量自定义函数。这也是通常意义上所讲的 UDF。而用户自定义聚集函数（UDA）与标量自定义函数不同，它是对多行进行调用返回单行结果。通常情况下，UDA 与 GROUP BY 语句结合将一个很大的结果集聚合成一个很小的结果集，甚至对整个表进行值汇总得到一行记录。

通过使用 CREATE AGGREGATE FUNTION 语句创建 UDA。这个语句包含 INIT_FN，UPDATE_FN、MERGE_FN、FINALIZE_FN 和 INERMEDIATE 选项。

这些*_FN 选项代表了函数处理的不同阶段：

- 初始化：INIT_FN 指定的函数进行初始化成员变量类似的初始化操作。如果我们省略这个选项，将使用默认的（no-op）函数。
- 更新：UPDATE_FN 指定的函数将在执行 GROUP BY 之前，针对原始结果集中的每行调用一次。该函数不同的实例对 GROUP BY 返回的不同的值进行调用。传递给这个函数的最终参数是一个指向基于原始值和第一参数值的更新值的指针。
- 合并：MERGE_FN 指定的函数将被调用任意次数，用于处理在 Impala 并行读取和处理数据时不同节点或不同线程产生的中间结果。传递给这个函数的最终参数是一个指向基于原始值和第一参数值的更新值的指针。

- 释放：FINALIZE_FN 指定的函数用于释放 UDF 请求的资源，比如释放内存，关闭文件句柄等。如果忽略这个选项，将使用默认函数（no-op）。

如果我们使用的是与底层函数一致的命名约定，Impala 可以自动确定依据第一个子句确定名称。

使用注意点：

（1）我们可以使用 Java 或 C++来编写 UDF。C++ UDF 是 Impala 新支持的，使用 C++的 UDF 能够充分使用本地代码库，提高性能。基于 Java 的 UDF 同时兼容 Impala 和 Hive。这使我们可以在 Impala 中重用 Hive 的 UDF。

（2）UDF 的函数体可以是.so 或者.jar 文件。这个文件必须存储在 HDFS 上，并使用 CREATE FUNCTION 语句将其分发到 Impala 的每一个节点上。

（3）在 SQL 赋值期间，Impala 将多次调用底层代码，处理的结果集有多少行就调用多少次。所有的 UDF 被认为是确定性的，也就意味着只要我们给它传相同的参数值，它就总是会返回相同的结果。如果 Impala 知道从之前的调用中返回的结果值，那么 Impala 可能会跳过这个调用，但是也可能会重新执行这个调用。因此在 Impala 查询执行过程中，对于计算出得结果不要依赖于 UDF 执行的次数，不用基于像当前时间、随机函数、外部数据源是否更新等外部变化的因素返回不用的结果。

（4）UDF 的参数名称包括他们的数量、位置和数据类型。

（5）我们可以使用不同的函数签名重载这个函数。从安全角度考虑，我们不能使用内嵌相同名称函数的 UDF。

（6）在 UDF 代码中，我们可以把函数返回值称为一个结构体。这个结构体包含两个字段，第一个字段是布尔值，用来标识这个值是否为 NULL，第二个值具有与函数返回值相同的类型。当函数返回值不为 NULL 时，将使用第二个值存放函数的返回值。

（7）Impala 目前不支持 ALTER FUNCTION 语句。所以如果想要重命名函数，将函数移动到另一个数据库，或者改变函数的签名或者其他属性，唯一的方式就是通过 DROP FUNCTION 语句删除当前的函数，再通过 CREATE FUNCTION 重新创建函数。

（8）UDF 与一个特定的数据库关联，所以在执行 CREATE FUNCTION 语句时，需要使用 USE 语句指定当前数据库或者直接以 db_name.function_name 的方式指定函数对应的数据库名称。

4.6.6 CREATE TABLE

创建一个表及指定表列的语法如下：

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
[(col_name data_type [COMMENT 'col_comment'], ...)]
[COMMENT 'table_comment']
[PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
[
```

```
  [ROW FORMAT row_format] [STORED AS file_format]
  ]
  [LOCATION 'hdfs_path']
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
data_type
  : primitive_type
primitive_type
  : TINYINT
  | SMALLINT
  | INT
  | BIGINT
  | BOOLEAN
  | FLOAT
  | DOUBLE
  | STRING
  | TIMESTAMP
row_format
  : DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
  [LINES TERMINATED BY 'char']
file_format:
  PARQUET | PARQUETFILE
  | TEXTFILE
  | SEQUENCEFILE
  | RCFILE
```

内部表和外部表：

默认情况下，Impala 创建的是内部表，Impala 负责管理表底层的数据文件。当我们删除表的时候，Impala 自动的删除对应的物理文件。如果我们指定 EXTERNAL 子句，Impala 将创建一张外部表。外部表对应的数据文件并非由 Impala 创建。外部表创建语句只是将 Impala 的表定义指向已存在的数据文件。当我们删除外部表时，Impala 不会删除数据文件。

分区表：

PARTITIONED BY 子句依据一列或者多列的值将数据文件分开存放。Impala 查询可以根据分区元数据只从磁盘读取需要的数据文件，尤其对于 JOIN 查询，将大大减少网络传输的数据量。

指定文件格式：

STORED AS 子句指定底层存储的数据文件的格式。目前，Impala 可以查询的文件格式的类型比它可以 CREATE/INSERT 的类型要多。使用 Hive 可以执行的创建和数据加载操作，在 Impala

中并不支持。比如：Impala 可以创建基于 SequenceFile 的表，但是不支持通过 INSERT 向其中插入数据。另外，Impala 还包含一些过程专门实现对不同类型的文件格式进行压缩。

默认情况下，Impala 使用以 CTRL-A 为分隔符的文件作为默认文件格式。通过 ROW FORMAT 子句可以使用其他字符作为列分隔符或者行结束符。我们常用的分隔符，如'\t'为 tab 键，'\n'为回车键，'\r'为换行键。

ESCAPED BY 子句可以应用于一个 Impala 创建的、使用 INSERT 语句插入数据的 TEXTFILE 格式的表，也可以应用于从其他位置直接拷贝到 Impala 表目录中的数据文件。选择一个在文件中从未出现过的字符作为转义字符，并把它放在字段内分隔符实例之前。在引号之内的字段值不利于 Impala 使用内嵌分隔符解析它，引号被认为是列值的一部分。如果我们想使用\作为转义字符，那么在 impala-shell 中需要指定 ESCAPED BY '\\'子句。

克隆表：

如果我们需要创建与某张表具有相同列、相同注释或者其他属性的一张表，可以使用如下语法：

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
LIKE [db_name.]table_name
[COMMENT 'table_comment']
[STORED AS file_format]
[LOCATION 'hdfs_path']
```

当我们使用 CREATE TABLE … LIKE 语法克隆一张表的表结构时，新的表使用与原始表相同的文件格式。如果我们想使用不同的文件格式存储，需要指定 STORED AS 子句。

Impala 不能直接创建 HBase 表，但是可以使用 CREATE TABLE … LIKE 语法克隆 HBase 表的结构。克隆的表将保留原始表的文件格式和元数据。

在 Avro 表上使用 CREATE TABLE … LIKE 语句时，存在一些特殊情况。比如：我们在一个无列的 Avro 表上使用 CREATE TABLE … LIKE 语句。一般情况下，我们可以先测试在 Hive 中，CREATE TABLE … LIKE 语句是否可以正常工作，如果不能，那么基本上也不能在 Impala 中正常使用。

如果原始表是分区表，那么新创建的表也将继承同样的分区列。由于新表是分区表，所以不会生成与原始表同样的真实的分区。在新表中需要通过插入数据或者 ALTER TABLE … ADD PARTITION 语句创建分区。

由于 CREATE TABLE … LIKE 只是操作表的元数据，而不是表中存的数据。如果要往表中插入数据，需要使用 INSERT INTO 从原始表中拷贝数据，如果新的表使用了不同的文件格式，拷贝数据的过程可以完成对文件的格式的转换。

```
CREATE TABLE AS SELECT:
```

使用该语句可以实现根据原始表的列定义一张新的表，同时将原始包对应的数据插入到这张

新表中。使用这个语句的优势在于在创建表时实现了数据的插入，不需要使用单独的 INSERT 语句来插入数据。这种创建表的写法有一个与关系型数据库中相同的缩写，叫做“CTAS”。具体的语法如下：

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] db_name.]table_name
[COMMENT 'table_comment']
[STORED AS file_format]
[LOCATION 'hdfs_path']
AS
select_statement
```

新创建的表将继承原始表的列名称。当然，我们也可以在 SELECT 语句中使用列别名来替换原始表的列名。使用这个语句不会继承原始表的表和列的注释。

示例：

```
[hadoop-cs1:21000] > select * from t1;
Query: select * from t1

Returned 0 row(s) in 0.17s
[hadoop-cs1:21000] > desc t1;
Query: describe t1
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| y    | int    |         |
| s    | string |         |
+------+--------+---------+
Returned 3 row(s) in 0.01s
[hadoop-cs1:21000] > insert into t1 values(1,2,'jcq0'),(3,4,'cjq0'),(5,6,'smon');
Query: insert into t1 values(1,2,'jcq0'),(3,4,'cjq0'),(5,6,'smon')
Inserted 3 rows in 0.40s
[hadoop-cs1:21000] > CREATE TABLE clone_of_t1 AS SELECT * FROM t1;
Query: create TABLE clone_of_t1 AS SELECT * FROM t1
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+
Returned 1 row(s) in 1.05s
```

```
[hadoop-cs1:21000] > CREATE TABLE empty_clone_of_t1 AS SELECT * FROM t1 WHERE 1=0;
Query: create TABLE empty_clone_of_t1 AS SELECT * FROM t1 WHERE 1=0
+-------------------+
| summary           |
+-------------------+
| Inserted 0 row(s) |
+-------------------+
Returned 1 row(s) in 0.58s
[hadoop-cs1:21000] > CREATE TABLE subset_of_t1 AS SELECT * FROM t1 WHERE x=1;
Query: create TABLE subset_of_t1 AS SELECT * FROM t1 WHERE x=1
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
Returned 1 row(s) in 0.80s
[hadoop-cs1:21000] > CREATE TABLE parquet_t1 like  t1 stored as parquet;
Query: create TABLE parquet_t1 like  t1 stored as parquet

Returned 0 row(s) in 0.08s
```

作为 CTAS 的一个附件功能，它可以实现数据文件的格式转换（目前支持 TEXTFILE、PARQUET）。但是我们没有办法为 TEXTFILE 表指定像分隔符之类的更低级别的属性。虽然我们可以对一个分区表使用 CTAS 语句，但是也无法为新表指定分区的属性。

可见性和元数据：

我们可以通过对一个表的 TBLPROPERTIES 子句指定元数据的属性。这个语句将一个键值对的列表存储在元数据库中。我们也可以通过 ALTER TABLE 语句修改表的属性。目前，Impala 在查询时不使用 TBLPROPERTIES 中的数据。但是如果要执行与其他的 Hadoop 组件进行交互 DDL 操作，比如创建 Avro 表或者 HBase 表时，就需要指定 TBLPROPERTIES 属性。

我们可以通过对一个表 WITH SERDEPROPERTIES 指定键值对，来指定表的 SerDes 属性。Impala 不会使用这些元数据，但是他们将用来构建支持的文件格式的序列化和反序列化器。某些属性值是为了与 Hive 在文件格式上保持兼容性变化。

如果要查看使用 CREATE TABLE … LIKE 或者 CREATE TABLE … AS SELECT 语句创建的表的列定义，可以使用 DESCRIBE table_name 格式的语句。如果要了解像数据文件位置，ROW FORMAT 或 STORED AS 对应的值这些详细的信息，可以使用 DESCRIBE FORMATTED table_name。这个语句还可以看到表的注释信息。

基于 Hive 考虑：

Impala 查询可以使用像表的总行数、单列不同值的个数等元数据信息。在 Impala 1.2.2 之前，可以使用 Hive 中的 ANALYZE TABLE 语句搜集统计信息。在 Impala 1.2.2 及更高的版本，不再需要通过 Hive 搜集统计信息，只需要使用新的 COMPUTE STATS 语句即可。

4.6.7　CREATE VIEW

创建视图的语句让我们可以简写缩写一个复杂的查询。如果一个查询语句包括 JOIN、表达式、列别名和其他 SQL 特性，那么它将让查询变的难于理解和维护。而创建视图，正好可以解决这个问题。

由于视图是一个逻辑结构体，不包含物理数据，所以 ALTER VIEW 语句只会改变元数据信息，而不会对 HDFS 上的数据文件进行任何操作。

```
CREATE VIEW view_name [(column_list)]
AS select_statement
```

语句类型：DDL

使用注意点：

CREATE VIEW 语句可以应用在如下场景：

（1）将复杂冗长的 SQL 语句缩写。我们可以通过应用程序、脚本、交互式查询对视图执行简单查询，来屏蔽视图本身的复杂性。

```
select * from view_name;
select * from view_name order by c1 desc limit 10;
```

越是复杂难度的查询，使用视图简化查询的效果越好。

（2）隐藏底层的表名和列名。简化由于基表名称变化带来的维护成本。如果基表的表名或者列名发生变化，我们只需要使用新的表名或者列名重建视图，基于视图的所有查询即可正常运行。

（3）让优化过的查询语句迅速的在所有的应用中起作用。如果我们发现针对 WHERE 条件，连接顺序，hints 或者其他一些技术能够让某一类查询变快，我们可以针对这些优化过的查询建立视图。应用程序通过相对简单的视图进行查询，而不用一遍一遍的调用复杂的优化逻辑。如果我们在视图运行之后找到了更好的查询优化方式，那么我们只需要用优化后的语句重建这些视图，就可以让应用程序中所有调用到这个视图的查询速度变快。

（4）为了简化某一类查询语句，尤其是那些包含了多表关联，列中包含了复杂表达式，或者其他 SQL 语法的复杂查询非常难于理解和调试。比如：我们可以创建一个包含了多表关联，WHERE 过滤条件以及查询需要的相关列的视图。有了这样一个视图，应用程序只需要对这个视图进行简单的 LIMIT、ORDER BY 等处理即可得到需要的结果。

4.6.8 DESCRIBE

这个语句用于显示表中得列名，列数据类型等元数据信息。语法如下：

```
DESCRIBE [FORMATTED] table
```

这里的 DESCRIBE 可以简写成 DESC。

DESCRIBE FORMATTED 可以显示更多的信息，显示的格式与 Hive 显示的类似。显示的信息包括表是内部表还是外部表，什么时候创建的，数据文件的格式，在 HDFS 上的位置，这个对象的类型是表还是视图等更底层的信息。

使用注意点：

在 impalad 进程重启之后，由于表上的元数据要在表被查询之前加载到各节点上，所以表上的第一个查询花的时间要稍微长一些。这种针对每个表会有的一次性的延迟，会使很多基准测试的结果不甚准确，甚至误导使用者。为了避免这种延迟的产生，可以对每张表使用 DESCRIBE 语句将其元数据提前做缓存。

当我们处理 HDFS 上的数据文件时，像 Impala 表数据文件所在的路径，namenode 的主机名称等信息就显的尤为重要。在 DESCRIBE FORMATTED 输出中，我们可以获得这些信息。我们可以在 LOAD DATA 或者 CREATE/ALTER TABLE 的 LOCATION 子句中使用 HDFS 的 URI 路径。另外，我们在拷贝、重命名数据文件时，也会在 Linux 操作系统命令中使用到 HDFS 的 URI 路径。

每个表都包括表相关的统计信息，列相关的统计信息。使用 SHOW TABLE STATS table_name 和 SHOW COLUMN STATS table_name 可以看到不同类别的信息。

示例：

如下示例显示了 DESCRIBE 和 DISCRIBE FORMATTED 的不同结果。

（1）DESCRIBE 对于表或者视图返回每列的名称、类型、注释。对于视图，如果列值是通过表达式运算出来的，那么列名将会自动生成形如_c0、_c1 等列名称。

（2）在 DESCRIBE FORMATTED 输出中，视图的表类型为 VIRTUAL_VIEW。因为视图的很多属性都是继承基表，所以很多属性显示为 NULL 或者空白。另外，在 DESCRIBE FORMATTED 输出中还可以查看到视图的定义。

（3）使用了 CREATE TABLE 不同选项创建的表，它的 DESCRIBE FORMATTED 输出也是不同的。T2 使用 CREATE EXTERNAL TABLE 语法创建，所以在 DESCRIBE FORMATTED 的输出中包含了 EXTERNAL_TABLE 关键字，另外 InputFormat 和 OutputFormat 字段表示了表的输入和输出的格式。

```
[hadoop-cs1:21000] > desc t1;
Query: describe t1
+------+--------+---------+
| name | type   | comment |
```

```
+------+--------+---------+
| x    | int    |         |
| y    | int    |         |
| s    | string |         |
+------+--------+---------+
Returned 3 row(s) in 0.11s
[hadoop-cs1:21000] > desc formatted t1;
Query: describe formatted t1
+-----------------------------+--------------------------------------------------------------+---------------------+
| name                        | type                                                         |
comment              |
+-----------------------------+--------------------------------------------------------------+---------------------+
|  #  col_name                                                             |  data_type
| comment              |
|                             | NULL                                                         |
NULL                 |
| x                           | int                                                          |
None                 |
| y                           | int                                                          |
None                 |
| s                           | string                                                       |
None                 |
|                             | NULL                                                         |
NULL                 |
|       #       Detailed       Table       Information       |       NULL
| NULL                 |
|   Database:                                                              |   default
| NULL                 |
|   Owner:                                                                   |   impala
| NULL                 |
|  CreateTime:                              |  Thu  Nov  20  17:43:32  CST  2014
| NULL                 |
|   LastAccessTime:                                                        |   UNKNOWN
| NULL                 |
|   Protect    Mode:                                                           |    None
| NULL                 |
|   Retention:                                                                      |   0
```

```
| NULL                 |
   | Location:                    | hdfs://hadoop-cm1:8020/user/hive/warehouse/t1
| NULL                 |
   |  Table  Type:                                          | MANAGED_TABLE
| NULL                 |
   |   Table   Parameters:                                        |   NULL
| NULL                 |
   |                    |numRows                                              |
0                    |
   |                                                  | transient_lastDdlTime
| 1416476987           |
   |                    | NULL                                               |
NULL                 |
   |   #    Storage    Information                                   |    NULL
| NULL                 |
   |   SerDe   Library:                                                       |
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe      | NULL              |
   | InputFormat:                | org.apache.hadoop.mapred.TextInputFormat
| NULL                 |
   |   OutputFormat:                                                          |
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL              |
   |  Compressed:                                                    |   No
| NULL                 |
   |  Num   Buckets:                                                   |   0
| NULL                 |
   |  Bucket   Columns:                                               |   []
| NULL                 |
   |  Sort   Columns:                                                 |   []
| NULL                 |
   +-----------------------------+---------------------------------------------
--------------+---------------------+
   Returned 27 row(s) in 0.03s
```

4.6.9 DROP DATABASE

从系统中删除数据库，从 HDFS 上删除关联的*.db 目录。为了避免丢失数据，在执行删除操作之前必须确实数据库是否为空。

语法：

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name;
```

语句类型：DDL

使用注意点：

在删除数据库之前，建议使用 DROP TABLE/VIEW 命令删除不需要的表和视图，使用 ALTER TABLE/VIEW 将需要保存的表和视图移动到其他数据库。

4.6.10　DROP FUNCTION

在 Impala 中包含 UDF 的 SELECT/INSERT 语句执行期间，无法对其执行删除操作。

语法：

```
DROP [AGGREGATE] FUNCTION [IF EXISTS] [db_name.]function_name
```

语句类型：DDL

4.6.11　DROP TABLE

删除 Impala 表。对于内部表，会级联删除 HDFS 上的数据文件；对于外部表，不会对数据文件做任何操作。

语法：

```
DROP TABLE [IF EXISTS] [db_name.]table_name
```

语句类型：DDL

使用注意点：

默认情况下，Impala 会级联删除表对应的数据文件和 HDFS 上的目录。如果我们执行了 DROP TABLE，但是数据文件没有被级联删除，可能有以下原因：

（1）如果表是外部表，Impala 不会对数据文件和目录做任何操作。当数据隶属于 Hadoop 的其他组件时，我们通常使用外部表。此时 Impala 只是对存储在原有位置的数据文件进行查询操作。

（2）Impala 可能无意保留下了这些数据文件，如果在 HDFS 中没有可用的位置，数据可能会被保存在 impala 用户的回收站中。

在删除表时，确保我们在正确的数据库中。建议删除表之前先使用 USE 语句切换当前数据库，或者使用 db_name.table_name 的形式指定表名。

IF EXISTS 子句之后，无论表是否存在，删除语句都不会报错。如果表存在，表将会被删除；如果表不存在，这个语句不产生任何影响。这个子句在执行标准的安装脚本，删除已经存在的对象或者创建新对象时非常有用。通过将 IF EXISTS 与 DROP 语句结合使用，或者将 IF NOT EXISTS 与 CREATE 语句结合使用，我们可以让一个脚本可以重复执行。

4.6.12 DROP VIEW

删除使用 CREATE VIEW 创建的指定的视图。因为视图是一个逻辑结构，不包含任何物理数据，所以删除视图操作只会影响元数据库中的元数据，HDFS 上的真实数据不会有任何变化。

语法：

```
DROP VIEW [database_name.]view_name
```

语句类型：DDL

4.6.13 EXPLAIN

该语句返回一个语句的执行计划。它从底层显示 Impala 如何读取数据，如何在各节点之间协调工作，传输中间结果，并获得最终结果的全过程。EXPLAIN 后面接一个完整的 SELECT 语句。

语法：

```
EXPLAIN { select_query | ctas_stmt | insert_stmt }
```

select_query 是一个 SELECT 语句；insert_stmt 是一个向表中插入或者重写数据的 INSERT 语句，具体可以使用 INSERT … SELECT 或者 INSERT … VALUES 语法。ctas_stmt 语句是一个使用了 AS SELECT 子句的 CREATE TABLE 语句。

使用注意点：

根据执行计划，我们可以判断这个语句的执行效率，如果发现你效率过低，可以调整语句本身，或者调整表结构。例如，我们可以通过改变 WHERE 的查询条件，对 JOIN 查询使用 hints，引入子查询，改变表的连接顺序，改变表的分区，搜集表及列统计信息或者其他的方式来对语句进行性能优化。

自底向上读取执行计划：

（1）执行计划的最后一部分内容是一些底层的详细信息。这些信息可以显示的是像需要被读取的数据量，根据这些信息我们可以判断分区的策略是否有效，还可以基于表的总大小及每个节点要读取的大小来判断这个语句需要花费多长时间。

（2）接下来我们可以看到哪些操作被分布到每个 Impala 节点上并行执行。

（3）从宏观上，我们还可以看到中间结果是如何组合起来在节点之间传输的。

我们一定要时刻记住，Impala 是用来优化数据仓库中超大表的全表扫描的。数据的结构和分布完全与 OLTP 系统不同，对一个大表的全表扫描是常态操作。当然，我们也可以通过减少需要扫描的数据的数量级来优化查询。比如，我们只扫描查询中需要的数据所在的分区，而不是扫描整张表来将查询的时间从分钟级提高到秒级。

扩展的 EXPLAIN 输出：

对于复杂查询的性能调整和容量规划，我们可以让 EXPLAIN 语句输出更详细的信息。在

impala-shell 中，执行 SET EXPLAIN_LEVEL=level,这里的 level 可以是 0 到 3 的整数，分别代表 minimal，standard，extended，和 verbose。

示例：

标准的 EXPLAIN 输出从物理级别到逻辑级别做了完整的显示。查询语句开始于扫描特定数量的数据；每个节点执行一个聚集操作（在本地节点执行 count(*)操作生成中间结果；中间结果被传输回协调者节点）；最后，将中间结果汇总为最终结果。

```
[hadoop-cs1:21000] > explain select count(*) from t1;
Query: explain select count(*) from t1
+-----------------------------------------------------------+
| Explain String                                          |
+-----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=42.00MB VCores=1 |
|                                                        |
| 03:AGGREGATE [MERGE FINALIZE]                           |
| |  output: sum(count(*))                                |
| |                                                      |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |                                                      |
| 01:AGGREGATE                                           |
| |  output: count(*)                                    |
| |                                                      |
| 00:SCAN HDFS [default.t1]                              |
|    partitions=1/1 size=27B                             |
+-----------------------------------------------------------+
Returned 12 row(s) in 0.04s
```

扩展 EXPLAIN 输出依据 COMPUTE STATS 搜集的统计信息显示了更为准确，详细的输出。默认的扩展输出中，像数据大小、分布情况等信息都显示为“unavailable”。Impala 可以确定裸数据文件的大小，但是在没有进行额外的分析之前为无法确定记录数，每列非重复值的个数等信息。COMPUTE STATS 语句执行分析之后，EXPLAIN 能够显示出为优化查询提供帮助的更为详细的信息。

```
[hadoop-cs1:21000] > set explain_level=extended;
EXPLAIN_LEVEL set to extended
[hadoop-cs1:21000] > explain select * from t1;
Query: explain select * from t1
+-----------------------------------------------------------+
| Explain String                                          |
```

```
+------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=32.00MB VCores=1 |
|                                                  |
| 01:EXCHANGE [PARTITION=UNPARTITIONED]                  |
| |  hosts=3 per-host-mem=unavailable                   |
| |  tuple-ids=0 row-size=28B cardinality=3               |
| |                                                   |
| 00:SCAN HDFS [default.t1, PARTITION=RANDOM]             |
|    partitions=1/1 size=27B                          |
|    table stats: 3 rows total                         |
|    column stats: all                              |
|    hosts=3 per-host-mem=32.00MB                       |
|    tuple-ids=0 row-size=28B cardinality=3              |
+------------------------------------------------------------+
Returned 12 row(s) in 0.01s
[hadoop-cs1:21000] >
```

4.6.14 INSERT

Impala 支持向通过 CREATE TABLE 创建的表或者通过 Hive 创建的表或分区中插入数据。Impala 当前支持：

- 通过 INSERT INTO 向表中添加数据。
- 通过 INSERT OVERWRITE 替换表中的数据。
- 从其他表中使用 SELECT 拷贝数据。在 Impala 1.2.1 或者更高的版本中，我们可以把 CREATE TABLE 和 INSERT 语句简化为单个 CREATE TABLE AS SELECT 操作。
- 在 INSERT 之前使用 WITH 子句，可以定义一个 SELECT 子查询。
- 使用带 VALUES 子句的常量表达式可以创建一行或者多行新记录。
- 通过 INSERT 指定插入列的名称和顺序。
- 在 INSERT …SELECT 向一个 Parquet 分区表插入数据的语句中，SELECT 关键字之前的 hint 可以很好的优化插入操作。这个 hint 包括[SHUFFLE] [NOSHUFFLE]。由于向 Parquet 分区表插入数据会引起同时向 HDFS 写多个文件，所以 INSERT 操作是一个资源敏感的操作。

语句类型：DML（但是仍然会影响 SYNC_DDL 选项）

使用注意点：

当我们插入一个表达式结果，尤其是存在内嵌的函数调用时，插入的数字列（像 INT、SMALLINT、TINYINT、FLOAT 等）需要使用 CAST()表达式将值转成特定的类型。Impala 不会自动的将一个大的值转换成小值。例如：如果我们要将一个余弦值插入 FLOAT 列，需要在插入

语句中显示的指定 CAST(COS(angle) AS FLOAT)。

任何向 Parquet 表插入数据的操作都需要 HDFS 文件系统上至少要有一个块大小的空闲空间。因为 Parquet 数据文件默认使用 1GB 作为一个块的默认大小。如果 HDFS 上没有至少一个块大小的空闲空间，将导致插入操作失败。

示例：

下面的示例使用同样的表定义创建了不同数据文件格式的新表，并向各表中插入数据。

```
[hadoop-cs1:21000] > CREATE DATABASE IF NOT EXISTS file_formats;
Query: create DATABASE IF NOT EXISTS file_formats

Returned 0 row(s) in 0.18s
[hadoop-cs1:21000] > USE file_formats;
Query: use file_formats
[hadoop-cs1:21000] > DROP TABLE IF EXISTS text_table;
Query: drop TABLE IF EXISTS text_table
[hadoop-cs1:21000] > CREATE TABLE text_table
                   > ( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
                   > STORED AS TEXTFILE;
Query: create TABLE text_table ( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3
TIMESTAMP ) STORED AS TEXTFILE

Returned 0 row(s) in 0.21s
[hadoop-cs1:21000] > DROP TABLE IF EXISTS parquet_table;
Query: drop TABLE IF EXISTS parquet_table
[hadoop-cs1:21000] > CREATE TABLE parquet_table
                   > ( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
                   > STORED AS PARQUET;
Query: create TABLE parquet_table ( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3
TIMESTAMP ) STORED AS PARQUET

Returned 0 row(s) in 0.10s
[hadoop-cs1:21000] >
```

使用 INSERT INTO TABLE 语法，可以将数据添加到已经存在数据的表中。对于那些有小量持续变化的数据，或者依据变化数据产生的批处理结果，使用 INSERT INTO TABLE 是最好的插入方式。

```
[hadoop-cs1:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.41s
```

```
[hadoop-cs1:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.46s
[hadoop-cs1:21000] > select count(*) from text_table;
+----------+
| count(*) |
+----------+
| 10 |
+----------+
Returned 1 row(s) in 0.26s
```

使用 INSERT OVERWRITE TABLE 语法，每个新插入的数据集将会替换表中已存在的数据。在数据仓库典型场景中，我们按照天、月份、季度加载数据，在加载时我们通常将上一个周期的数据删除。有的时候，我们可能需要保留整个数据集，然后转换处理某些数据做更复杂的分析。

如下示例中，我们通过 INSERT INTO 插入了 5 条记录，然后使用 INSERT OVERWRITE 替换的方式插入了 3 条记录。在最终的插入之后，这个表只包含 3 条记录。

```
[hadoop-cs1:21000] > insert into table parquet_table select * from default.tab1;
Inserted 5 rows in 0.35s
[hadoop-cs1:21000] > insert overwrite table parquet_table select * from
default.tab1
limit 3;
Inserted 3 rows in 0.43s
[hadoop-cs1:21000] > select count(*) from parquet_table;
+----------+
| count(*) |
+----------+
| 3 |
+----------+
Returned 1 row(s) in 0.43s
```

VALUES 语句可以指定常量值向表中插入一条或者多条记录。插入时，插入的值、类型、表达式的顺序必须和表定义严格一致。

如下示例显示了如何将不同类型的表达式、常量、函数返回值插入表中.

```
[hadoop-cs1:21000] > create table val_test_1 (c1 int, c2 float, c3 string, c4 boolean,
c5 timestamp);
Query: create table val_test_1 (c1 int, c2 float, c3 string, c4 boolean, c5
timestamp)

Returned 0 row(s) in 0.12s
```

```
[hadoop-cs1:21000] > insert into val_test_1 values (100, 99.9/10, 'abc', true, now());
Query: insert into val_test_1 values (100, 99.9/10, 'abc', true, now())
ERROR: AnalysisException: Possible loss of precision for target table 'file_formats.val_test_1'.
Expression '99.9 / 10.0' (type: DOUBLE) would need to be cast to FLOAT for column 'c2'
[hadoop-cs1:21000] > create table val_test_2 (id int, token string);
Query: create table val_test_2 (id int, token string)

Returned 0 row(s) in 0.19s
[hadoop-cs1:21000] > insert overwrite val_test_2 values (1, 'a'), (2, 'b'), (-1,'xyzzy');
Query: insert overwrite val_test_2 values (1, 'a'), (2, 'b'), (-1,'xyzzy')
Inserted 3 rows in 1.19s
```

如下示例显示了，如果尝试将数据插入 Impala 当前不支持的数据格式的表时，将会报“not implemented”错误：

```
DROP TABLE IF EXISTS sequence_table;
CREATE TABLE sequence_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS SEQUENCEFILE;
DROP TABLE IF EXISTS rc_table;
CREATE TABLE rc_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS RCFILE;
[hadoop-cs1:21000] > insert into table rc_table select * from default.tab1;
Remote error
Backend 0:RC_FILE not implemented.
[hadoop-cs1:21000] > insert into table sequence_table select * from default.tab1;
Remote error
Backend 0:SEQUENCE_FILE not implemented.
```

向分区表中插入数据需要在插入语句中指定分区列:

```
[hadoop-cs1:21000] > create table t1 (i int) partitioned by (x int, y string);
Query: create table t1 (i int) partitioned by (x int, y string)

Returned 0 row(s) in 0.24s
[hadoop-cs1:21000] > insert into t1 partition(x=10, y='a') select id from t2;
```

```
Query: insert into t1 partition(x=10, y='a') select id from t2
Inserted 3 rows in 1.74s
[hadoop-cs1:21000] >
```

我们可以将一个表的所有列的数据拷贝至另一张表。另外，我们也可以拷贝一些列，或者指定列到其他表中。

并发考虑：

每个 INSERT 操作都使用唯一名称创建一个数据文件，因此我们可以同时运行多个 INSERT INTO 语句而不会引起文件名冲突。在数据插入期间，数据被临时存放在数据目录的子目录中，在这期间，我们无法在 Hive 中对表执行查询。如果插入操作失败，临时的数据文件和子目录不会自动删除。我们需要使用 hdfs dfs -rm –r 命令后面跟上工作子目录的全路径，以删除临时子目录和文件。

VALUES 语法：

使用 VALUES 语法，我们可以将带有具体列值的行插入到表中。如下示例说明：

（1）如何插入单行记录。
（2）如何插入多行记录。
（3）使用 INSERT OVERWRITE 替换现有内容。
（4）VALUES 中得值可以是常量，函数结果或者表达式。

```
[hadoop-cs1:21000] > create table val_example(id int,col_1 boolean,col_2 double);
Query: create table val_example(id int,col_1 boolean,col_2 double)

Returned 0 row(s) in 0.20s
[hadoop-cs1:21000] > desc val_example;
Query: describe val_example
+-------+---------+---------+
| name  | type    | comment |
+-------+---------+---------+
| id    | int     |         |
| col_1 | boolean |         |
| col_2 | double  |         |
+-------+---------+---------+
Returned 3 row(s) in 1.08s
[hadoop-cs1:21000] > insert into val_example values (1,true,100.0);
Query: insert into val_example values (1,true,100.0)
Inserted 1 rows in 0.40s
[hadoop-cs1:21000] > select * from val_example;
```

```
Query: select * from val_example
+----+-------+-------+
| id | col_1 | col_2 |
+----+-------+-------+
| 1  | true  | 100   |
+----+-------+-------+
Returned 1 row(s) in 0.32s
[hadoop-cs1:21000] > insert overwrite val_example values (10,false,pow(2,5)),
                   > (50,true,10/3);
Query: insert overwrite val_example values (10,false,pow(2,5)), (50,true,10/3)
Inserted 2 rows in 0.40s
[hadoop-cs1:21000] > select * from val_example;
Query: select * from val_example
+----+-------+-------------------+
| id | col_1 | col_2             |
+----+-------+-------------------+
| 10 | false | 32                |
| 50 | true  | 3.333333333333333 |
+----+-------+-------------------+
Returned 2 row(s) in 0.33s
[hadoop-cs1:21000] >
```

使用 INSERT … VALUES…语句时，不支持使用列的子集或者使用与现有列不同的顺序。VALUES 值必须使用与表定义相同的列顺序，对于不想插入数据的列，可以指定为 NULL。

为了在其他的语句中使用 VALUES 子句中的值，可以将这些值使用 AS 语句作为一张拥有别名的表来使用：

```
[hadoop-cs1:21000] > select * from (values(4,5,6),(7,8,9)) as t;
Query: select * from (values(4,5,6),(7,8,9)) as t
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 4 | 5 | 6 |
| 7 | 8 | 9 |
+---+---+---+
Returned 2 row(s) in 0.20s
[hadoop-cs1:21000] > select * from (values(1 as c1, true as c2, 'abc' as
c3),(100,false,'xyz')) as t;
Query: select * from (values(1 as c1, true as c2, 'abc' as c3),(100,false,'xyz'))
as t
```

```
+-----+-------+-----+
| c1  | c2    | c3  |
+-----+-------+-----+
| 1   | true  | abc |
| 100 | false | xyz |
+-----+-------+-----+
Returned 2 row(s) in 0.10s
```

我们甚至可以根据例子中的方式利用常量或者函数返回值创建一张小表用于和其他的表关联或者 UNION ALL。

HBase 的考虑：

我们可以针对如下 HBase 表使用 INSERT 语句：

（1）我们可以使用 INSERT … VALUES 语法向 HBase 表中插入单行或者少量行的记录。通过这种方式向 HBase 表中插入数据是使用 Impala 插入数据的最佳方式，因为插入到 HBase 中的数据不会因为插入少量的数据而引起碎片。

（2）我们可以通过 INSERT…SELECT 语句向 HBase 表中插入任意行数的记录。

（3）如果我们插入 HBase 表中的记录含有相同的 HBase 键值，那么只有最后插入的记录的值是对 Impala 可见的。我们也可以根据这个原理，使用 INSERT … VALUES 语句插入与原键值相同的记录完成对 HBase 中记录的更新。因为同样的原因，我们通过 INSERT … SELECT 插入到 HBase 表中的记录数可能由于键值重复导致真正看到的数据比事实的记录数要少。

（4）我们不能使用 INSERT OVERWRITE 向 HBase 表中插入记录。对于 HBase 表来说，新的记录不能覆盖原有记录，只能以增加的方式插入数据。

（5）当我们创建映射到 HBase 的 Impala 或者 Hive 的表时，我们在 INSERT 中指定的列的顺序可能和 CREATE TABLE 语句中定义的不同。在底层，HBase 基于列族的方式管理列。这可能引起插入操作的不匹配，尤其是使用 INSERT INTO hbase_table SELECT * FROM hdfs_table 的方式插入数据时。在插入数据之前，我们需要先通过 DESCRIBE 语句查看表中列的顺序，根据这个顺序调整 INSERT… SELECT 中 SELECT 列表中的列顺序。

4.6.15 INVALIDATE METADATA

该语句用于将表的元数据标记为过期。在一张表通过 Hive 命令行创建后，我们执行该命令后，才能让 Impala 对该表正常访问。Imapla 在对一个元数据标记为过期的表进行查询之前，会自动重新加载最新的相关元数据信息。同时，与 REFRESH 相比，这个操作是一个成本极高的操作，因为 REFRESH 是对元数据进行增量更新。在一般情况下，比如向已存在的表添加数据文件之类的操作，建议优先选用 REFRESH。

为了准确的响应查询，Impala 客户端必须能够直接访问数据库和表最新的元数据信息。因此，如果 Impala 更新了某些 Impala 和 Hive 共享的对象的定义信息，由 Impala 缓存过的原来的元数据

信息必须被更新为最新的。当然，这也并不是说要更新所有的元数据信息，Impala 只会更新需要的元数据信息。

INVALIDATE METADATA 和 REFRESH 对比：

INVALIDATE METADATA 在执行查询需要访问元数据信息时加载相关表的所有元数据信息。这个操作尤其是对那些拥有很多分区的大表来说，成本相当高。REFRESH 加载元数据速度更快，因为只加载新增加的数据文件的块位置数据，所以成本也相对较低。如果数据被像 HDFS balancer 这样的外部方式更改，使用 INVALIDATE METADATA 可以避免降低本地读取的性能损失。

INVALIDATE METADATA 语法如下：

```
INVALIDATE METADATA [table_name]
```

默认情况下，缓存的所有表的元数据信息都会被清除。如果在语法中指定了表名，那么将会清除指定表的元数据信息。即使是对单张表，INVALIDATE METADATA 也比 REFRESH 成本更高。在向表中添加数据文件之后，优先使用 REFRESH 操作。

在以下情况下，impalad 实例会请求元数据更新操作：

- 元数据发生变化
- 其他 impalad 实例或者 Hive 修改了元数据
- Impala shell 或者通过 ODBC 方式修改了数据库

如果在某个节点上执行了 ALTER TABLE、INSERT 或者其他的修改语句之后，在同一个节点上执行查询时，不需要进行元数据更新操作。

数据库或者表的元数据修改包括：

- 通过 Hive 执行 ALTER、CREATE、DROP 或 INSERT 操作
- 通过 impalad 执行 CREATE TABLE、ALTER TABLE 或 INSERT 操作

INVALIDATE METADATA 语句会强制表的元数据信息过期，这样，在下次表被引用的时候元数据信息就会重新加载。对于一个超大表进行 INVALIDATE METADATA 将会消耗大量的时间。如果使用 REFRESH，可能可以避免不可预知的延迟。

下面的例子演示了可以对一个通过 Hive 创建的新的表（SequenceFile 或 HBase 表）使用 INVALIDATE METADATA 语句。在 INVALIDATE METADATA 语句执行之前，如果我们想引用一张表，Impala 可能会报“table not found”错误。DESCRIBE 语句将会使更新该表的最新的元数据信息，这也会避免下次对这些表进行查询时由于重新加载元数据信息带来的延迟。

```
[hadoop-cs1:21000] > invalidate metadata;
Query: invalidate metadata

Returned 0 row(s) in 1.08s
[hadoop-cs1:21000] > describe t1;
```

```
Query: describe t1
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| i    | int    |         |
| x    | int    |         |
| y    | string |         |
+------+--------+---------+
Returned 3 row(s) in 0.01s
[hadoop-cs1:21000] > describe t2;
Query: describe t2
+------+------+---------+
| name | type | comment |
+------+------+---------+
| id   | int  |         |
+------+------+---------+
Returned 1 row(s) in 0.01s
[hadoop-cs1:21000] >
```

如果我们启动 impala-shell 时要确保所有访问的表的元数据保持最新，需要再运行 impala-shell 时使用-r，或者-refresh_after_connect 选项。因为这个选项为每个查询增加了刷新元数据信息的操作，对于那些有很多分区的超大表或者每天更新数据的表不建议使用这个选项。

示例：

示例演示了在 Hive 中创建一个新的数据库和一张新的表，在 Impala 中使用表全名进行 INVALIDATE METADATA 操作，之后可以发现 Impala 就可以识别新的数据库和新的表了。这一操作是 Impala 1.2.4 的新功能，在早期的版本中，无法为特定的表进行 INVALIDATE METADATA 操作。

```
-bash-4.1$ hive
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.input.dir.recursive is deprecated. Instead, use mapreduce.input.fileinputformat.input.dir.recursive
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.max.split.size is deprecated. Instead, use mapreduce.input.fileinputformat.split.maxsize
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.min.split.size is deprecated. Instead, use mapreduce.input.fileinputformat.split.minsize
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.min.split.size.per.rack is deprecated. Instead, use mapreduce.input.fileinputformat.split.minsize.per.rack
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.min.split.size.per.node is deprecated. Instead, use mapreduce.input.fileinputformat.split.minsize.per.node
```

```
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use mapreduce.job.reduces
14/11/20 18:42:20 INFO Configuration.deprecation: mapred.reduce.tasks.speculative.execution is deprecated. Instead, use mapreduce.reduce.speculative
14/11/20 18:42:21 WARN conf.HiveConf: DEPRECATED: Configuration property hive.metastore.local no longer has any effect. Make sure to provide a valid value for hive.metastore.uris if you are connecting to a remote metastore.

Logging initialized using configuration in jar:file:/usr/lib/hive/lib/hive-common-0.12.0-cdh5.0.2.jar!/hive-log4j.properties
hive> create database new_db_from_hive;
OK
Time taken: 0.786 seconds
hive> create table new_db_from_hive.new_table_from_hive (x int);
OK
Time taken: 0.249 seconds
hive> quit;
-bash-4.1$
-bash-4.1$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > show databases like 'new*';
Query: show databases like 'new*'

Returned 0 row(s) in 0.01s
[hadoop-cs1:21000] > refresh new_db_from_hive.new_table_from_hive;
Query: refresh new_db_from_hive.new_table_from_hive
ERROR: AnalysisException: Database does not exist: new_db_from_hive
[hadoop-cs1:21000] > invalidate metadata new_db_from_hive.new_table_from_hive;
Query: invalidate metadata new_db_from_hive.new_table_from_hive
```

```
Returned 0 row(s) in 0.74s
[hadoop-cs1:21000] > show databases like 'new*';
Query: show databases like 'new*'
+------------------+
| name             |
+------------------+
| new_db_from_hive |
+------------------+
Returned 1 row(s) in 0.01s
[hadoop-cs1:21000] >  show tables in new_db_from_hive;
Query: show tables in new_db_from_hive
+---------------------+
| name                |
+---------------------+
| new_table_from_hive |
+---------------------+
Returned 1 row(s) in 0.11s
[hadoop-cs1:21000] >
```

4.6.16 LOAD DATA

LOAD DATA 语句通过将数据文件从一个目录移动到另一个目录简化了对 Impala 表的 ETL 过程。

语法：

```
LOAD DATA INPATH 'hdfs_file_or_directory_path' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]
```

语句类型：

DML（仍然影响 SYNC_DDL 选项）

注意事项：

加载的数据文件是被移动而不是被拷贝到 Impala 数据目录的。

我们可以指定移动 HDFS 路径中的单个文件或者所有文件。我们无法使用通配符只移动某些需要的文件。在加载一个目录下所有的数据文件时，需要保证数据文件位于该目录之下，而不是嵌套的子目录之下。

目前，Impala 的 LOAD DATA 只能从 HDFS 加载数据文件，而不能从本地文件系统加载数据文件。目前不支持 Hive 中得 LOCAL 关键字。我们必须指定一个路径，而不是形如 hdfs://的 URI。

为了保证加载的速度，加载过程对错误的检查很有限。如果加载的数据文件的文件格式错误，

源文件中得列数与目标表中得列数不一致，或者其他类型的不匹配问题，Impala 在 LOAD DATA 时不会报任何错误。但是加载完毕之后，在对表进行查询时，Impala 会报运行时错误或者其他不可预知的错误。目前，Impala 唯一做的检查就是避免未压缩的文本文件和 LZO 方式压缩的文本文件出现在同一张表中。

当我们在 LOAD DATA 中指定一个 HDFS 目录时，目录中得任何隐藏文件（文件的名字以.开头的文件）将会被忽略。

除非有命名冲突，否则加载到目标位置的数据文件与源数据文件名称保持一致。

通过 LOAD DATA 这种方式将 HDFS 中的文件加载到 Impala 中，简化了加载的过程，同时我们也不必关注表对应的数据文件究竟是在哪个 HDFS 目录中，如何分布的，分别对应哪个数据库，哪张表。我们可以通过 DESCRIBE FORMATTED table_name 语句，快速地了解数据文件和表的对应关系。

PARTITION 子句对于增加新的数据分区特别方便。当我们生成了按时间段，或者地理位置，或者其他特定含义的新的数据时，可以将数据直接加载到 Impala 数据目录，将其作为 Impala 的一个或者多个分区。对于分区表，加载数据时，必须使用常量值指定要将数据加载到那个分区。

示例：

首先我们使用一个简单的 shell 脚本生成几个包含数字串的文件，将文件上传到 HDFS 上。

```
[root@localhost jcq0]# cat random_strings.sh
#!/bin/sh
num=$1;
for (( c=1; c<=num; c++ ))
do
   echo $RANDOM;
done;
[root@localhost jcq0]# chmod +x random_strings.sh
[root@localhost jcq0]# sh random_strings.sh 5
11507
4156
32039
15247
10826
$ random_strings.sh 1000 | hdfs dfs -put - /user/cloudera/thousand_strings.txt
$ random_strings.sh 100 | hdfs dfs -put - /user/cloudera/hundred_strings.txt
$ random_strings.sh 10 | hdfs dfs -put - /user/cloudera/ten_strings.txt
```

然后，我们创建表，并将数据加载进去。大家需要留意，除非指定了 STORE AS 子句，Impala 默认使用 TEXTFILE 做默认的数据文件格式，使用 Ctrl-A（16 进制的 01）作为分隔符。本例中使用的是只有一个列的表，所以分隔符不太重要。对于大型的 ETL 作业，我们需要使用像 Parquet

或者 Avro 这样的数据文件格式，并将数据加载到相应的文件格式的 Impala 表中。

```
[hadoop-cs1:21000] > create table t1 (s string);
[hadoop-cs1:21000] > load data inpath '/user/cloudera/thousand_strings.txt' into
table
t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary |
+----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 1 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.61s
[hadoop-cs1:21000] > select count(*) from t1;
Query finished, fetching results ...
+------+
| _c0 |
+------+
| 1000 |
+------+
Returned 1 row(s) in 0.67s
[hadoop-cs1:21000] > load data inpath '/user/cloudera/thousand_strings.txt' into
table
t1;
ERROR: AnalysisException: INPATH location '/user/cloudera/thousand_strings.txt'
does
not exist.
```

在数据加载进表之后，通过上例中得最后一条语句可以看到，数据文件已经被移动而不是复制到了 Impala 数据目录中。

下面的示例中，我们可以看到数据文件被移动到 Impala 数据目录中后，保持了原来的数据文件名称。

```
$ hdfs dfs -ls /user/hive/warehouse/load_data_testing.db/t1
Found 1 items
-rw-r--r-- 1 cloudera cloudera 13926 2014-11-20 18:45
/user/hive/warehouse/load_data_testing.db/t1/thousand_strings.txt
```

下面的示例将演示在 INTO TABLE 和 OVERWRITE INTO TABLE 之间的不同之处。表原始数据有 1000 行。在执行了带有 INTO TABLE 子句的 LOAD DATA 语句后，表中的数据多了 100

行，总共 1100 行。在执行了带有 OVERWRITE INTO TABLE 子句的 LOAD DATA 语句加载了 10 条数据后，原有的数据丢失了，表中总共包含 10 条数据。

```
    [hadoop-cs1:21000] > load data inpath '/user/cloudera/hundred_strings.txt' into
table
    t1;
    Query finished, fetching results ...
    +----------------------------------------------------------+
    | summary |
    +----------------------------------------------------------+
    | Loaded 1 file(s). Total files in destination location: 2 |
    +----------------------------------------------------------+
    Returned 1 row(s) in 0.24s
    [hadoop-cs1:21000] > select count(*) from t1;
    Query finished, fetching results ...
    +------+
    | _c0 |
    +------+
    | 1100 |
    +------+
    Returned 1 row(s) in 0.55s
    [hadoop-cs1:21000] > load data inpath '/user/cloudera/ten_strings.txt' overwrite
into
    table t1;
    Query finished, fetching results ...
    +----------------------------------------------------------+
    | summary |
    +----------------------------------------------------------+
    | Loaded 1 file(s). Total files in destination location: 1 |
    +----------------------------------------------------------+
    Returned 1 row(s) in 0.26s
    [hadoop-cs1:21000] > select count(*) from t1;
    Query finished, fetching results ...
    +-----+
    | _c0 |
    +-----+
    | 10 |
    +-----+
    Returned 1 row(s) in 0.62s
```

4.6.17 REFRESH

为了能够准确的对数据进行查询，我们通过 imapla-shell、JDBC 或者 ODBC 所连接到的节点将充当协调者节点，它负责获取将要查询的数据库和表的最新元数据信息。

使用 REFRESH 语句加载最新的元数据信息，特定表的块位置信息等的操作适用于如下场景：

- 在加载了新的数据文件到表对应的 HDFS 目录之后。
- 在 Hive 中执行了 ALTER TABLE、INSERT、LOAD DATA 或者其他对表的修改的 SQL 语句之后。

我们只需要在协调者节点上执行 REFRESH 语句即可。协调者节点负责依据本节点上的元数据信息将任务分发到集群所有的 Impala 节点上，而不依赖于其他节点的元数据信息。

REFRESH 负责从元数据库中重新加载表的元数据信息，而且是增量的加载新的数据文件的块位置信息。这是一个可以对单表进行的低负载操作，通常在向表添加了新的数据文件之后，将使用这个命令更新元数据信息。

REFRESH 语法：

```
REFRESH table_name
```

这个命令只更新指定表的元数据。这张表不管是 Impala 使用 CREATE TABLE 方式创建的表，还是之前使用 INVALIDATE METADATA 语句加载过全量的元数据信息，都必须保证这张表可以被 Impala 识别。

REFRESH 与 INVALIDATE METADATA 的区别已在介绍 INVALIDATE METADATA 部分提及，在此不再赘述。

示例：

这里演示的是在手动向 Impala 数据目录添加了数据文件之后，使用 REFRESH 刷新元数据信息的例子。

```
[hadoop-cs1:21000] > refresh t1;
Query: refresh t1

Returned 0 row(s) in 0.18s
[hadoop-cs1:21000] > refresh t2;
Query: refresh t2

Returned 0 row(s) in 0.24s
[hadoop-cs1:21000] > select * from t1;
Query: select * from t1
+---+---+------+
| x | y | s    |
```

```
+---+---+------+
| 1 | 2 | jcq0 |
| 3 | 4 | cjq0 |
| 5 | 6 | smon |
+---+---+------+
Returned 3 row(s) in 0.30s
[hadoop-cs1:21000] > select * from t2;
Query: select * from t2

Returned 0 row(s) in 0.19s
[hadoop-cs1:21000] >
```

4.6.18　SELECT

SELECT 语句执行查询，从一张或者多张表获取数据，并返回由行列组成的结果集。在 Impala 的 INSERT 语句中，通常会包含一个 SELECT 语句用于定义要从源表中拷贝哪些数据。

1. Impala 的 SELECT 查询支持：

- SQL 数据类型：BOOLEAN、TINYINT、SMALLINT、INT、BIGINT、FLOAT、DOUBLE、TIMESTAMP、STRING。
- 在 SELECT 关键字前指定 WITH 子句中可以定义一个子查询，在 SELECT 中可以引用这个子查询。
- DISTINCT 子句。
- 在 FROM 子句中使用子查询。
- WHERE、GROUP BY、HAVING 子句。
- ORDER BY 在排序的同时也可以使用 LIMIT 关键字。
- 包括 LEFT、RIGHT、SEMI、FULL、OUTER 等各种 JOIN 连接查询。在 Impala 1.2.2 及以上版本支持 CROSS JOIN 连接。在性能调整期间，我们可以通过在 SELECT 关键字后加 STRAIGHT_JOIN 覆盖原有的连接顺序。
- UNION ALL。
- LIMIT。
- 外部表。
- 像>、<、=的关系操作符。
- 像+、-的数学运算符。
- 逻辑/布尔运算符 AND、OR、NOT 等，但是不支持&& 、||、!等。
- 像 COUNT、SUM、CAST、LIKE、IN、BETWEEN、COALESCE 等通用的 SQL 内嵌函数。

2. JOIN

连接查询对两个或者多个表的数据进行某种形式的合并，并返回包含特定要求的结果集。

语法：

```
SELECT select_list FROM
table_or_subquery1 [INNER] JOIN table_or_subquery2 |
table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} JOIN
table_or_subquery2 |
table_or_subquery1 LEFT SEMI JOIN table_or_subquery2
[ ON col1 = col2 [AND col3 = col4 ...] |
USING (col1 [, col2 ...]) ]
[other_join_clause ...]
[ WHERE where_clauses ]

SELECT select_list FROM
table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
[other_join_clause ...]
WHERE
col1 = col2 [AND col3 = col4 ...]

SELECT select_list FROM
table_or_subquery1 CROSS JOIN table_or_subquery2
[other_join_clause ...]
[ WHERE where_clauses ]
```

3. SQL-92 和 SQL-89 连接

查询中显式指定 JOIN 关键字的是 SQL-92 风格的 JOIN。使用 ON 或者 USING 关键字指定哪些列作为连接键。

```
SELECT t1.c1, t2.c2 FROM t1 JOIN t2
ON t1.id = t2.id and t1.type_flag = t2.type_flag
WHERE t1.c1 > 100;

SELECT t1.c1, t2.c2 FROM t1 JOIN t2
USING (id, type_flag)
WHERE t1.c1 > 100;
```

即使两个表关联列的列名不同，也可以使用 ON 关键字进行连接。而 USING 是 ON 的缩写形式，专门用在各表关联列具有相同名称的情况下。当然，我们也可以不使用 USING，ON 关键字，直接使用 WHERE 条件进行关联列的等值比较，但是这种书写方式会将连接比较和过滤混淆，降低了可读性，不易维护。

使用逗号分隔参与连接的各表或者子查询的查询是 SQL-89 风格的 JOIN。在这种查询中，使用的就是 WHERE 条件对关联列进行等值比较。这种语法很容易使用，但是也很容易由于删除某些 WHERE 子句导致连接无法正常工作。

```
SELECT t1.c1, t2.c2 FROM t1, t2
WHERE
t1.id = t2.id AND t1.type_flag = t2.type_flag
AND t1.c1 > 100;
```

4. 自连接

Impala 支持自连接，允许对某张表的不同列进行关联查询以展示数据之间的父子关系或者树形结构。对于自连接查询无须显式指定自连接关键字。只要对同一张表指定不同的别名，将其作为两张表对待进行连接即可。

```
SELECT t1.id, t2.parent, t1.c1, t2.c2 FROM a t1,a t2
 WHERE t1.id= t2.parent;
```

5. 笛卡尔连接

为了避免由于书写错误产生海量结果集，Impala 不支持诸如下述例子中得笛卡尔连接。

```
SELECT ... FROM t1 JOIN t2;
SELECT ... FROM t1, t2;
```

如果我们想实现笛卡尔乘积，可以使用 CROSS JOIN 作为连接操作符。但是 CROSS JOIN 连接形式不支持 ON 子句，因为该连接将对连接的表的所有行进行组合。通过这种方式产生的结果集仍然可以使用 WHERE 子句进行过滤操作。

```
SELECT ... FROM t1 CROSS JOIN t2;
SELECT ... FROM t1 CROSS JOIN t2 WHERE tests_on_non_join_columns;
```

6. 内连接和外连接

内连接是最常用最熟悉的连接类型：内连接的结果集包含所有参与连接的表中匹配的列，这些列具有满足在不同表之间关联列的等值匹配。如果参与连接的表具有相同的列名，则需要使用指定完全限定名或者列别名对该列进行引用。Impala 的内连接支持 SQL-89 SQL-92 语法。

```
SELECT t1.id, c1, c2 FROM t1, t2 WHERE t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 JOIN t2 ON t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

外连接从左手型表，或右手型表，或者全外连接表获取所有的行数据。如果外连接的表中没有与其他表关联匹配的数据，在结果集中相关列会被置为 NULL。在一个外连接查询中需要包含

OUTER 关键字作为连接操作符，OUTER 可与 LEFT、RIGHT、FULL 配合使用。

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id;
```

对于外连接查询，Impala 使用的是 SQL-92 语法，也就是说使用 JOIN 关键字而不是使用逗号对参与关联的表进行连接。Impala 不支持(+)或者*=等兼容 SQL-89 的连接语法。

7. 等值连接和非等值连接

默认情况下，Impala 对表进行等值连接查询。无论是 INNER、OUTER、FULL、SEMI 连接，都执行的是等值连接。

在 Impala 1.2.2 或者更高版本中，可以使用比较运算符实现非等值连接。这种类型的连接可以避免产生超出资源限制的超大结果集。如果我们执行的非等值连接产生的结果集大小可以接受，那么我们可以使用 CROSS JOIN 操作符，并且在 WHERE 子句中进行额外的比较操作。

```
SELECT ... FROM t1 CROSS JOIN t2 WHERE t1.total > t2.maximum_price;
```

8. 半连接

半连接相对较少使用。如果使用的是 LEFT SEMI JOIN，就意味着只有左手型表中满足 ON 或者 WHERE 子句与指定的与右手型表关联条件的数据会被返回。无论左手型表中的一行与右手型表中得多少行匹配，只会有单行的实例被返回。

```
SELECT t1.c1, t1.c2, t1.c2 FROM t1 LEFT SEMI JOIN t2 ON t1.id = t2.id;
```

Impala 不支持自然连接和反连接。

我们可以在如下情况下使用连接查询：

（1）当需要从不同的物理上独立存储的表进行关联获取数据时。比如，我们可以对包含地址信息的商业数据与电话列表数据或者人口普查数据关联进行交叉检验。

（2）将数据归一化，连接查询允许我们减少数据复制，将不同的数据存储在不同的表中。这种技术最常用的方式是在传统的关系型数据库中。例如，为了减少在多个表中的像客户名称这样长字符串的重复存储，我们可以为每个客户名称指定一个客户 ID 号。这样客户名称只需要存储一次，如果我们需要展现客户名称列，只需要将其他表的客户 ID 列与客户表的客户 ID 列进行关联就可以了。

（3）对于那些很少使用的某些列，我们可以将其移动到其他表中以减少大部分查询的负载。比如在员工表中，biography 的字段可能很少使用，那么我们可以将这个字段放在单独的表中以减少对员工表大部分查询的负载。对于需要 biography 列的查询，我们可以通过将该表与员工表关联获得。

当在 ON 或者 WHERE 子句中的关联列具有相同名称时，我们需要指定像 db_name.table_name

这样的完全限定名，或者表别名，列别名来减少歧义，但同时也增加了复杂性，减低了可读性。

```
select t1.c1 as first_id, t2.c2 as second_id from
t1 join t2 on first_id = second_id;

select fact.custno, dimension.custno from
customer_data as fact join customer_address as dimension
using (custno)
```

为了控制连接查询的结果集，我们需要在 ON 或者 USING 子句中指定表的列名，或者在 WHERE 条件中对列进行等值比较：

```
[hadoop-cs1:21000] > select c_last_name, ca_city from customer join
customer_address
where c_customer_sk = ca_address_sk;
+-------------+-----------------+
| c_last_name | ca_city |
+-------------+-----------------+
| Lewis | Fairfield |
| Moses | Fairview |
| Hamilton | Pleasant Valley |
| White | Oak Ridge |
| Moran | Glendale |
...
| Richards | Lakewood |
| Day | Lebanon |
| Painter | Oak Hill |
| Bentley | Greenfield |
| Jones | Stringtown |
+-------------+------------------+
Returned 50000 row(s) in 9.82s
```

JOIN 连接的风险在于一个低效的查询可能消耗过多的资源。Impala 对 JOIN 进行限制来解决这一问题。为了减少查询失控的机会，Impala 要求每个 JOIN 查询至少包含一个等式谓词。例如：表 T1 有 1000 行，T2 有 1000000 行记录，查询 SELECT columns FROM t1 JOIN t2 将返回 10 亿条记录（1000*1000000）。Impala 需要查询指定一个等式谓词，在这个查询中可以是 ON t1.c1 = t2.c2 或者 t1.c1 = t2.c2 。

即使指定了等式谓词，最终生成的结果集仍然很大。我们可以使用 LIMIT 谓词返回结果集的子集。

```
[hadoop-cs1:21000] > select c_last_name, ca_city from customer, customer_address
```

```
where
   c_customer_sk = ca_address_sk limit 10;
   +-------------+-----------------+
   | c_last_name | ca_city |
   +-------------+-----------------+
   | Lewis | Fairfield |
   | Moses | Fairview |
   | Hamilton | Pleasant Valley |
   | White | Oak Ridge |
   | Moran | Glendale |
   | Sharp | Lakeview |
   | Wiles | Farmington |
   | Shipman | Union |
   | Gilbert | New Hope |
   | Brunson | Martinsville |
   +-------------+-----------------+
   Returned 10 row(s) in 0.63s
```

或者我们可以使用比较操作符或者聚集函数，将一个大的结果集压缩成一个更小的结果集：

```
   [hadoop-cs1:21000] > select distinct c_last_name from customer, customer_address
where
   c_customer_sk = ca_address_sk
   and ca_city = "Green Acres";
   +-------------+
   | c_last_name |
   +-------------+
   | Hensley |
   | Pearson |
   | Mayer |
   | Montgomery |
   | Ricks |
   ...
   | Barrett |
   | Price |
   | Hill |
   | Hansen |
   | Meeks |
   +-------------+
   Returned 332 row(s) in 0.97s
```

```
[hadoop-cs1:21000] > select count(distinct c_last_name) from customer, customer_address
where c_customer_sk = ca_address_sk
and ca_city = "Green Acres"
and substr(c_last_name,1,1) = "A";
+-----------------------------+
| count(distinct c_last_name) |
+-----------------------------+
| 12 |
+-----------------------------+
Returned 1 row(s) in 1.00s
```

由于 JOIN 查询可能涉及读取大量的磁盘数据，跨网络发送大量数据，并加载大量数据到内存中做的比较和过滤，我们可以做基准测试、性能分析来优化查询，以找到适合我们的数据集最有效的连接方式、硬件容量、网络配置，以及集群负载。

Impala 支持两类 JOIN，一种是分区 JOIN，一种是广播 JOIN。如果表或者列的统计信息不准确，数据分布不均匀都可以引起 Impala 选择错误的执行方式，如果出现了这种情况，可以考虑使用 hints 作为临时的解决方案。

9. ORDER BY 子句

ORDER BY 子句对 SELECT 语句产生的结果集中的单列或者多列进行排序。对于分布式查询，这是一个代价非常高的操作，因为在排序之前，整个结果集需要传输到一个节点上进行排序。这可能比没有 ORDER BY 的语句需要更多的内存容量。即使对于有 ORDER BY 或者没有 ORDER BY 的子句的查询大概需要的时间差不多，但是主观上讲有 ORDER BY 子句的查询要更慢一些，因为只有所有的结果集都返回而且被处理完毕后才能计算出结果。

ORDER BY 语法如下：

```
ORDER BY col1 [, col2 ...] [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

默认的排序顺序为升序，也就是说比较小的结果集在开头，比较大的结果集在末尾。指定 DESC 关键字，可以将排序顺序指定为降序排列。

NULLS FIRST/NULLS LAST 子句可以控制 NULL 值究竟是位于结果集的开头，还是位于结果集的末尾。

Impala 执行的每一个包含 ORDER BY 子句的查询都可以使用 LIMIT 关键字。因为对一个超大的结果集进行排序需要消耗更多的内存，而 top-N 查询可以尽量减少协调者节点消耗更多的内存。我们可以指定 LIMIT 子句作为查询的一部分，或者在 impala-shell 中使用 SET DEFAULT_ORDER_BY_LIMIT=...对一个会话所有的查询设置默认的限制，也可以在启动 impalad 进程启动时指定-default_query_options default_order_by_limit=...在实例级别进行限制。

10. GROUP BY 子句

使用 GROUP BY 子句时需要使用像 COUNT()、SUM()、AVG()、MIN()、MAX()这样的聚集函数。在 GROUP BY 子句中指定的所有的列都不会参与聚集运算。

示例中的查询可以计算出的销售量最高的 5 项。因为条目 ID 不参与任何聚集运算，所以在 GROUP BY 中指定了条目 ID。

```
select
ss_item_sk as Item,
count(ss_item_sk) as Times_Purchased,
sum(ss_quantity) as Total_Quantity_Purchased
from store_sales
group by ss_item_sk
order by sum(ss_quantity) desc
limit 5;
+-------+-----------------+--------------------------+
| item | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
| 9325 | 372 | 19072 |
| 4279 | 357 | 18501 |
| 7507 | 371 | 18475 |
| 5953 | 369 | 18451 |
| 16753 | 375 | 18446 |
+-------+-----------------+--------------------------+
```

HAVING 子句可以过滤聚集函数计算的结果。例如，我们可以计算销售量在 100 以上的销售额最低的商品：

```
select
ss_item_sk as Item,
count(ss_item_sk) as Times_Purchased,
sum(ss_quantity) as Total_Quantity_Purchased
from store_sales
group by ss_item_sk
having times_purchased >= 100
order by sum(ss_quantity)
limit 5;
+-------+-----------------+--------------------------+
| item | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
```

```
| 13943 | 105 | 4087 |
| 2992 | 101 | 4176 |
| 4773 | 107 | 4204 |
| 14350 | 103 | 4260 |
| 11956 | 102 | 4275 |
+-------+----------------+------------------------+
```

当进行涉及科学或者财务的数据运算时，我们需要使用 FLOAT 或者 DOUBLE 类型的值来存储真实的浮点数，虽然它的精度很高，但是仍然可能无法准确的表示分数值。因此，如果对 FLOAT 或者 DOUBLE 列进行 GROUP BY 操作，得到的结果可能不能完全匹配原始文本数据中的数值。为了和数据中精确的值进行匹配，我们可以使用四舍五入操作、BETWEEN 操作符或者其他的数学运算来实现这种近似的操作。在如下示例中，ss_wholesale_cost 列返回的成本值接近但并不完全等于原始的成本值。

```
select ss_wholesale_cost, avg(ss_quantity * ss_sales_price) as
avg_revenue_per_sale
from sales
group by ss_wholesale_cost
order by avg_revenue_per_sale desc
limit 5;
+-------------------+----------------------+
| ss_wholesale_cost | avg_revenue_per_sale |
+-------------------+----------------------+
| 96.94000244140625 | 4454.351539300434 |
| 95.93000030517578 | 4423.119941283189 |
| 98.37999725341797 | 4332.516490316291 |
| 97.97000122070312 | 4330.480601655014 |
| 98.52999877929688 | 4291.316953108634 |
+-------------------+----------------------+
```

需要注意的是，这里的批发成本原本为 96.34、98.38 这样的小数，我们这里计算出的值由于精度的限制可能比实际的值偏大或者偏小一点点。使用 FLOAT 或者 DOUBLE 类型很容易引起数据精度的问题，所以金融行业的数据处理系统使用占用空间少，效率高的数据类型来存储货币，同时也避免了四舍五入的错误。

11. HAVING 子句

对带有 GROUP BY 子句的 SELECT 查询执行过滤操作，它过滤的是聚集函数运算的结果，而不是原始数据的行。

12. LIMIT 子句

LIMIT 子句限制 SELECT 产生的结果集输出的最大行数。在如下的情况下使用：

（1）TOP-N 查询，比如销售得最好的商品类别前 10 位，或者网站流量最大的 50 台机器的机器名称。

（2）从一个表或者查询中采样数据。对于一个没有 ORDER BY 子句限制的查询，使用 LIMIT 限制获取的数据是随机的。

（3）保持从海量结果集中返回的数据规模。如果一个查询由于 WHERE 子句匹配的行数比预期的多得多，或者其他不可预知的情况产生了一个很大的结果集，使用 LIMIT 可以有效地减少结果集的规模。

使用注意点：

在早期版本中，LIMIT 的值必须是一个数字常量。在 Impala 1.2.1 或者更高版本中，我们也可以使用一个数字表达式。

在 Impala 1.2.1 或者更高的版本中，我们可以使用带有 OFFSET 的 LIMIT 子句生成一个不同于 TOP-N 的查询，比如，我们要获取第 top-11 到第 top-20 的值。这项技术可以用来模拟分页的效果。因为 Impala 的查询一般都会消耗大量的 IO，尤其对于那些无法重写应用逻辑的情况下，使用这项技术是最佳选择。为了获得更好的性能和可扩展性，我们建议在应用端缓存查询的结果，并且在应用逻辑中限定返回的数据集的大小。

示例：

```
[hadoop-cs1:21000] > create database limits;
Query: create database limits

Returned 0 row(s) in 0.17s
[hadoop-cs1:21000] > use limits;
Query: use limits
[hadoop-cs1:21000] > create table numbers (x int);
Query: create table numbers (x int)

Returned 0 row(s) in 0.10s
[hadoop-cs1:21000] >  insert into numbers values (1), (3), (4), (5), (2);
Query: insert into numbers values (1), (3), (4), (5), (2)
Inserted 5 rows in 1.39s
[hadoop-cs1:21000] > select x from numbers limit 100;
Query: select x from numbers limit 100
+---+
| x |
```

```
+---+
| 1 |
| 3 |
| 4 |
| 5 |
| 2 |
+---+
Returned 5 row(s) in 0.29s
[hadoop-cs1:21000] > select x from numbers limit 3;
Query: select x from numbers limit 3
+---+
| x |
+---+
| 1 |
| 3 |
| 4 |
+---+
Returned 3 row(s) in 0.30s
[hadoop-cs1:21000] > select x from numbers where x > 2 limit 2;
Query: select x from numbers where x > 2 limit 2
+---+
| x |
+---+
| 3 |
| 4 |
+---+
Returned 2 row(s) in 0.30s
[hadoop-cs1:21000] >
```

对于 top-N 查询，我们使用 ORDER BY 和 LIMIT 结合使用。如果我们设置了 DEFAULT_ORDER_BY_LIMIT 选项，我们就不需要为每个 ORDER BY 查询指定 LIMIT 子句了。另外，我们还可以通过设置 ABORT_ON_DEFAULT_LIMIT_EXCEEDED 选项来避免 LIMIT 子句截断结果集。

```
[hadoop-cs1:21000] > select x from numbers order by x;
Query: select x from numbers order by x
ERROR: NotImplementedException: ORDER BY without LIMIT currently not supported
[hadoop-cs1:21000] > set default_order_by_limit=3;
DEFAULT_ORDER_BY_LIMIT set to 3
```

```
[hadoop-cs1:21000] > select x from numbers order by x;
Query: select x from numbers order by x
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.30s
[hadoop-cs1:21000] > set abort_on_default_limit_exceeded=true;
ABORT_ON_DEFAULT_LIMIT_EXCEEDED set to true
[hadoop-cs1:21000] > select x from numbers order by x;
Query: select x from numbers order by x
ERRORS: DEFAULT_ORDER_BY_LIMIT has been exceeded.


Backend 0:DEFAULT_ORDER_BY_LIMIT has been exceeded.

[hadoop-cs1:21000] > set default_order_by_limit=1000;
DEFAULT_ORDER_BY_LIMIT set to 1000
[hadoop-cs1:21000] > select x from numbers order by x;
Query: select x from numbers order by x
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
Returned 5 row(s) in 0.31s
[hadoop-cs1:21000] >
```

13. OFFSET 子句

OFFSET 子句可以查询自逻辑第一行之后的某行开始返回结果。返回的结果集的记录从 0 开始编号，所以如果是 OFFSET 0 相当于没有使用 OFFSET 子句。这个子句经常结合 ORDER BY 和 LIMIT 一起使用。

示例：

如下的例子演示了类似于传统数据库中的分页功能。因为 Impala 典型的查询场景是处理上 MB 或 GB 字节的数据，需要每次从磁盘读取大量的数据文件，所以这种从大量结果集中返回极小量数据的方式显得并不高效。只有针对很老的逻辑复杂的应用，我们使用单条语句重写应用代码，然后对缓存的结果集进行分页显示在页面上。

```
[hadoop-cs1:21000] > create table numbers (x int);
[hadoop-cs1:21000] > insert into numbers select x from very_long_sequence;
Inserted 1000000 rows in 1.34s
[hadoop-cs1:21000] > select x from numbers order by x limit 5 offset 0;
+----+
| x |
+----+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+----+
Returned 5 row(s) in 0.26s
[hadoop-cs1:21000] > select x from numbers order by x limit 5 offset 5;
+----+
| x |
+----+
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
+----+
Returned 5 row(s) in 0.23s
```

14. UNION 子句

UNION 子句可以合并多个查询的结果集。默认情况下，合并的结果集就好像是使用了 DISTINCT 操作符一样。

语法：

```
query_1 UNION [DISTINCT | ALL] query_2
```

使用注意点：

对结果集进行 UNION 就好像进行了 UNION DISTINCT 一样。但是同时 UNION 要对结果集进行去重操作，这是一个对内存非常敏感的操作。如果我们已经从业务上可以保证最终合并的结果集中没有重复的记录，或者业务上允许重复的记录出现，使用 UNION ALL 是一个更好的选择。

当我们对 UNION ALL/UNION 的数据进行 ORDER BY 的时候，通常会结合使用 LIMIT。如果我们设置了 DEFAULT_ORDER_BY_LIMIT 选项，为了让该设置对整个结果集生效，我们需要将 UNION 的查询放进一个子查询，在对子查询进行 SELECT 和 ORDER BY 操作。

示例：

首先我们生成一些数据，数据中包含多个 1 的重复值。

```
[hadoop-cs1:21000] > create table few_ints (x int);
Query: create table few_ints (x int)

Returned 0 row(s) in 0.21s
[hadoop-cs1:21000] > insert into few_ints values (1), (1), (2), (3);
Query: insert into few_ints values (1), (1), (2), (3)
Inserted 4 rows in 1.45s
[hadoop-cs1:21000] > set default_order_by_limit=1000;
DEFAULT_ORDER_BY_LIMIT set to 1000
```

这里演示了 UNION ALL 返回的所有记录是不会过滤重复值的。

```
[hadoop-cs1:21000] > select x from few_ints order by x;
Query: select x from few_ints order by x
+---+
| x |
+---+
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 4 row(s) in 0.31s
[hadoop-cs1:21000] > select x from few_ints union all select x from few_ints;
Query: select x from few_ints union all select x from few_ints
+---+
| x |
+---+
| 1 |
```

```
| 1 |
| 2 |
| 3 |
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 8 row(s) in 0.35s
[hadoop-cs1:21000] > select * from (select x from few_ints union all select x from
                  > few_ints) as t1 order by x;
Query: select * from (select x from few_ints union all select x from few_ints) as
t1 order by x
+---+
| x |
+---+
| 1 |
| 1 |
| 1 |
| 1 |
| 2 |
| 2 |
| 3 |
| 3 |
+---+
Returned 8 row(s) in 0.45s
[hadoop-cs1:21000] > select x from few_ints union all select 10;
Query: select x from few_ints union all select 10
+----+
| x  |
+----+
| 10 |
| 1  |
| 1  |
| 2  |
| 3  |
+----+
Returned 5 row(s) in 0.42s
[hadoop-cs1:21000] >
```

这个例子演示了如果使用没有 ALL 的 UNION 子句，结果集会进行去重操作，同时这个查询可能要消耗更多的内存，花费更多的时间。如果结果集大到有几百万或者几十亿条数据，Impala 不推荐使用 UNION 操作。

```
[hadoop-cs1:21000] > select x from few_ints union select x+1 from few_ints;
Query: select x from few_ints union select x+1 from few_ints
+---+
| x |
+---+
| 3 |
| 4 |
| 1 |
| 2 |
+---+
Returned 4 row(s) in 0.63s
[hadoop-cs1:21000] > select x from few_ints union select 10;
Query: select x from few_ints union select 10
+----+
| x  |
+----+
| 2  |
| 10 |
| 1  |
| 3  |
+----+
Returned 4 row(s) in 0.58s
[hadoop-cs1:21000] > select * from (select x from few_ints union select x from few_ints)
                   > as t1 order by x;
Query: select * from (select x from few_ints union select x from few_ints) as t1 order by x
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.63s
```

15. WITH 子句

这个子句可以放在 SELECT 语句之前，用于为复杂表达式定义一个别名，该别名定义好之后可以在后续的 SELECT 语句中多次引用。WITH 子句除了使用完后不能将 WITH 子句中定义的表和列名持久化存储之外，很像 VIEW，另外 WITH 子句定义的别名不能与已经存在的表或者视图的名称冲突。

我们可以使用子查询重写一个查询语句来代替 WITH 子句。但是 WITH 子句具有以下特点：

- 方便维护。尤其是在包含有 UNION、聚集函数等复杂表达式被重复引用的时候。
- 可读性强。将复杂的查询独立出来使 SQL 查询语句更容易理解。
- 提高了与其他传统的关系型数据库的兼容性，尤其是 Oracle 也支持类似的语法。

兼容性标准：SQL:1999

```
[hadoop-cs1:21000] > with t1 as (select 1), t2 as (select 2) select * from t1 union all select * from t2;
Query: with t1 as (select 1), t2 as (select 2) select * from t1 union all select * from t2
+---+
| 1 |
+---+
| 2 |
| 1 |
+---+
Returned 2 row(s) in 0.26s
```

16. Hints

Impala 的 SQL 支持 Hints，使用 Hints 可以更好的从底层调整 SQL 查询的工作方式。对于那些丢失了统计信息或者其他因素导致查询成本异常昂贵时，使用 Hints 可以作为临时的解决方案。Hints 的使用方式为使用[ ]将特定的 Hints 括起来放在 SQL 语句中。

[BROADCAST] 和 [SHUFFLE] 控制 JOIN 查询的连接方式，可以在语句中的 JOIN 关键字后面加上这个 Hints。

- [SHUFFLE]：JOIN 操作使用“分区”技术，将所有表的相关联的行使用 hash 算法分片，具有相同 hash 值的数据被发送到同一个节点上处理。这种方式对于处理大表之间的关联非常有效。
- [BROADCAST]：JOIN 操作使用“广播”技术，将右手型表的所有内容发送到与之关联的所有节点上。这种方式对于大表与小表的关联非常有效。当表或者索引的统计信息不可用时，这种连接方式是默认的连接方式。如果过期的元数据可能引起 Impala 选择错误的 JOIN 方式。

使用 EXPLAIN 可以确认一个特定的查询使用了什么样的关联策略。

使用 JOIN 将一个很大的 customer 表与一个很小的不到 100 行数据的 lookup 表进行关联。右手型的表将会被广播到所有节点上参与连接操作。我们也可以使用[broadcast]的 Hints 强制查询使用广播的连接方式。

```
select customer.address, state_lookup.state_name
from customer join [broadcast]state_lookup
on customer.state_id = state_lookup.state_id;
```

对于两个非常大的表使用[SHUFFLE]进行分区关联是最有效的连接方式。我们也可以通过测试的方式来找到每一个表与其他表关联最有效的关联策略。我们也可以使用[shuffle]的 Hints 强制使用分区连接的关联策略。

```
select weather.wind_velocity, geospatial.altitude
from weather join [shuffle]geospatial
on weather.lat = geospatial.lat and weather.long = geospatial.long;
```

对于三张或者三张以上的表参与的关联操作，Hints 对指定的 JOIN 关键字两侧的表有效。表关联是从左向右进行的。如下的例子中的表 t1 和 t2 采用分区连接，得到的结果集与表 t3 使用的是广播连接方式。

```
select t1.name, t2.id, t3.price
from t1 join [shuffle]t2 join [broadcast]t3
on t1.id = t2.id and t2.id = t3.id;
```

当我们向分区表插入数据时，尤其是向 Parquet 格式的表中插入数据时，我们可以对 INSERT 语句使用 Hints 来降低资源的使用，提高性能。需要注意的点如下：

（1）这个 Hints 仅对 Impala 1.2.2 及以上版本有效。

（2）我们可以在使用 INSERT 向分区表插入数据因为容量限制导致失败，或者插入性能低下时使用这些 Hints。

（3）使用[SHUFFLE]或者 [NOSHUFFLE]的 Hints 时，需要将其放在 PARTITION 关键字之后，SELECT 关键字之前。

（4）[SHUFFLE]关键字将导致选择并发向 HDFS 写文件数最小的执行计划，使用 1GB 的内存缓存不同分区的数据。这将大大减少 INSERT 操作的资源消耗，由于 Impala 总是将同一分区的数据存放在同一节点上，这一操作可能引起节点间的数据传输。

（5）[NOSHUFFLE]关键字将导致选择一个插入速度较快，但是会生成很多小数据文件的执行计划。这很可能引起 INSERT 操作由于超过容量限制而失败。

（6）如果 INSERT … SELECT 查询的源表中包含分区列，即使没有列统计信息，Impala 也会自动选择[SHFFULE]方式进行插入。

（7）如果 INSERT … SELECT 查询源表中的分区列的统计信息可用，Impala 将根据列的

distinct 值的个数和 INSERT 操作的节点数来确定使用[SHUFFLE]还是[NOSHUFFLE]。当然，我们也可是使用 Hints 来强制 Impala 选择我们想要的执行计划。

17. DISTINCT 操作符

DISTINCT 操作符使用在 SELECT 语句中，用于过滤并删除结果集中的重复值。

```
select distinct c_birth_country from customer;
```

该语句用来过滤单列的重复值。

```
select distinct c_salutation, c_last_name from customer;
```

该语句用来过滤两列组合的重复值。

我们可以使用 DISTINCT 结合聚集函数使用，比如可以和 COUNT()结合使用来计算一个列包含多少个不同值。

```
select count(distinct c_birth_country) from customer;
```

该语句用来计算单列非重复值的个数。

```
select count(distinct c_salutation, c_last_name) from customer;
```

该语句用来计算两个列非重复值的个数。

但是 Impala 不支持在同一个查询中使用多次聚集函数。例如，我们不能在一个查询的 SELECT 列表中查询 COUNT(DISTINCT c_first_name)和 COUNT(DISTINCT c_last_name)。

4.6.19　SHOW

SHOW 可以用来查看不同类型的 Impala 对象的相关信息。我们可以使用 SHOW object_type 语句用来查看当前数据库中所有的该类型的对象，也可以使用 SHOW object_typeIN database_name 语句来查看特定数据库中的所有该类型的对象。

语法：

显示特定类型的对象列表。

```
SHOW DATABASES [[LIKE] 'pattern']
SHOW SCHEMAS [[LIKE] 'pattern'] - an alias for SHOW DATABASES
SHOW TABLES [IN database_name] [[LIKE] 'pattern']
SHOW [AGGREGATE] FUNCTIONS [IN database_name] [[LIKE] 'pattern']
SHOW CREATE TABLE [database_name].table_name
SHOW TABLE STATS [database_name.]table_name
SHOW COLUMN STATS [database_name.]table_name
```

语法中的 pattern 参数可以是一个字符串常量，也可是使用通配符*代表任意字符，或者使用|

连接多个通配符，表示或的关系。这里的 LIKE 关键字也是可选的。由于所有的对象名称使用小写存储，所以匹配的字符串中也要使用小写字符。

```
[hadoop-cs1:21000] > show databases 'd*';
Query: show databases 'd*'
+---------+
| name    |
+---------+
| d1      |
| d2      |
| d3      |
| default |
+---------+
Returned 4 row(s) in 0.02s
[hadoop-cs1:21000] > show databases like 'd*';
Query: show databases like 'd*'
+---------+
| name    |
+---------+
| d1      |
| d2      |
| d3      |
| default |
+---------+
Returned 4 row(s) in 0.01s
[hadoop-cs1:21000] > show tables in default 't*';
Query: show tables in default 't*'
+------+
| name |
+------+
| t1   |
| t2   |
| tab1 |
| tab2 |
| tab3 |
| test |
+------+
Returned 6 row(s) in 0.01s
[hadoop-cs1:21000] > show tables in default like 't*';
```

```
Query: show tables in default like 't*'
+------+
| name |
+------+
| t1   |
| t2   |
| tab1 |
| tab2 |
| tab3 |
| test |
+------+
Returned 6 row(s) in 0.01s
[hadoop-cs1:21000] >
```

使用注意点：

在第一次连接到一个实例上之后，通常会使用 SHOW DATABASES 语句列出所有的数据库。然后使用 USE db_name 语句可以切换到指定的数据库，执行 SHOW TABLES 可以对该库中的表进行查看，并执行 SELECT 或 INSERT 等一系列的操作。

为了对执行 CREATE TABLE 创建的表定义进行变更，我们可能需要执行一系列的 ALTER TABLE 语句。为了确认后续的 ALTER TABLE 语句对表定义的影响，可以使用 SHOW CREATE TABLE 来显示当前表的 CREATE TABLE 表定义语句。使用 SHOW CREATE TABLE 生成的脚本可以很轻松的完成对当前表的克隆，而不需要再依次执行原始的 CREATE TABLE 语句和后续的 ALTER TABLE 语句。使用 SHOW CREATE TABLE 生成的脚本克隆时，需要将数据库名称，LOCATION 字段和其他一些变量的信息修改为与目标库匹配的信息。

SHOW TABLE STATS 和 SHOW COLUMN STATS 对于调整和诊断性能，尤其对大表和复杂连接查询来说非常重要。

默认情况下，SHOW FUNCTIONS 显示的是 UDFs，而 SHOW AGGREGATE FUNCTIONS 显示的是 UDAFs。SHOW FUNCTIONS 的输出包含了每个函数的参数签名。在 DROP FUNCTION 语句中可以指定参数签名删除特定数据类型参数的函数。

如果要显示 Impala 内嵌的函数，可以指定数据库名称为_impala_builtins。

```
[hadoop-cs1:21000] > show functions in _impala_builtins;
Query: show functions in _impala_builtins
+----------------+----------------------------------------+
| return type    | signature                              |
+----------------+----------------------------------------+
| BOOLEAN        | ifnull(BOOLEAN, BOOLEAN)               |
| TINYINT        | ifnull(TINYINT, TINYINT)               |
```

```
| SMALLINT      | ifnull(SMALLINT, SMALLINT)          |
| INT          | ifnull(INT, INT)                    |
...
[hadoop-cs1:21000] > show functions in _impala_builtins like '*week*';
Query: show functions in _impala_builtins like '*week*'
+-------------+----------------------------+
| return type | signature                  |
+-------------+----------------------------+
| INT         | weekofyear(TIMESTAMP)      |
| TIMESTAMP   | weeks_add(TIMESTAMP, INT)    |
| TIMESTAMP   | weeks_add(TIMESTAMP, BIGINT) |
| TIMESTAMP   | weeks_sub(TIMESTAMP, INT)    |
| TIMESTAMP   | weeks_sub(TIMESTAMP, BIGINT) |
| INT         | dayofweek(TIMESTAMP)       |
+-------------+----------------------------+
Returned 6 row(s) in 0.12s
```

SHOW DATABASES 输出包括了数据库_impala_builtins，在上面的例子中我们演示了可以在这个特殊的数据库中查看内嵌函数的定义。

如果启用了授权机制，那么使用 SHOW 只能输出你拥有权限的对象。而那些我们没有权限的数据库，表或者其他对象都会被隐藏。如果某个对象在数据库中存在，我们又无法通过 SHOW 命令看到这个对象，那我们就有理由怀疑是否在这个对象上具有访问的权限。

如下我们将演示如何从一个不熟悉的系统中找到一张特定的表。当我们初次连接上 Impala 默认的当前数据库名称为 DEFAULT，我们可以列出系统中存在的所有数据库。我们也可以使用 SHOW TABLES IN db_name 在不进入这个数据库的情况下列出该数据库中的所有表。当然这和在数据库内部使用 SHOW TABLES 的效果是一样的。

```
[hadoop-cs1:21000] > show databases;
+--------------------+
| name |
+--------------------+
| _impala_builtins |
| analyze_testing |
| avro |
| ctas |
| d1 |
| d2 |
| d3 |
| default |
```

```
| file_formats |
| hbase |
| load_data |
| partitioning |
| regexp_testing |
| reports |
| temporary |
+--------------------+
Returned 14 row(s) in 0.02s
[hadoop-cs1:21000] > show tables in file_formats;
+--------------------+
| name |
+--------------------+
| parquet_table |
| rcfile_table |
| sequencefile_table |
| textfile_table |
+--------------------+
Returned 4 row(s) in 0.01s
[hadoop-cs1:21000] > use file_formats;
[hadoop-cs1:21000] > show tables like '*parq*';
+--------------------+
| name |
+--------------------+
| parquet_table |
+--------------------+
Returned 1 row(s) in 0.01s
```

4.6.20　USE

默认情况下，当我们连接到 Impala 实例时，当前数据库为 default。在 impala-shell 回话中执行 USE db_name 可以切换到指定的数据库。在当前数据库中执行 CREATE TABLE、INSERT 、SELECT 或者其他语句都可以直接指定表名，而不需要指定数据库名称作为前缀。

使用注意点：

以下情况需要切换默认的数据库：

（1）为了避免每次引用表时指定数据库名称。执行 SELECT * FROM t1 JOIN t2，而不是 SELECT * FROM db.t1 JOIN db.t2。

（2）为了在同一个数据库中执行一系列的操作。比如执行创建表、插入数据、对表进行查询

等一系列的操作。

为了在启动 impala-shell 时直接自动连接到特定的数据库，可以在启动 impala-shell 时指定-d db_name 选项。-d 选项对于运行 SQL 脚本非常有用，使用这个选项之后就无须把 USE 语句硬编码在 SQL 脚本中。

4.7 内嵌函数

Impala 支持数据运算、字符串操作、日期运算等几类内嵌函数。内嵌函数可以使 SQL 返回经过格式化、运算、类型转换后的结果，而无须经过另外一个应用进行特别处理。通过在需要的位置使用函数调用，我们可以使 SQL 查询和过程中使用表达式一样方便。

Impala 支持的函数包括：

- 数学运算函数
- 类型转换函数
- 日期和时间函数
- 条件函数
- 字符串函数
- 聚集函数

我们可以直接通过 SELECT 语句调用这些函数。对于大多数的函数，我们可以忽略 FROM 子句，直接对常量进行运算。

```
[hadoop-cs1:21000] > select abs(-1);
Query: select abs(-1)
+-----------+
| abs(-1.0) |
+-----------+
| 1         |
+-----------+
Returned 1 row(s) in 0.63s
[hadoop-cs1:21000] > select concat('The rain ', 'in Spain');
Query: select concat('The rain ', 'in Spain')
+--------------------------------+
| concat('the rain ', 'in spain') |
+--------------------------------+
| The rain in Spain              |
+--------------------------------+
```

```
Returned 1 row(s) in 0.19s
[hadoop-cs1:21000] > select power(2,5);
Query: select power(2,5)
+-----------------+
| power(2.0, 5.0) |
+-----------------+
| 32              |
+-----------------+
Returned 1 row(s) in 0.09s
```

当我们使用 FROM 子句并把一列作为函数的输入参数时，这个函数将对结果集中的每条记录的相应列进行运算。

```
[hadoop-cs1:21000] > select s,concat(s,' is a name.') from t1;
Query: select s,concat(s,' is a name.') from t1
+------+------------------------+
| s    | concat(s, ' is a name.') |
+------+------------------------+
| jcq0 | jcq0 is a name.        |
| cjq0 | cjq0 is a name.        |
| smon | smon is a name.        |
+------+------------------------+
Returned 3 row(s) in 0.28s
```

如果任何函数的输入参数为 NULL，那么函数的输出结果也是 NULL。

```
[hadoop-cs1:21000] > select cos(null);
Query: select cos(null)
+-----------+
| cos(null) |
+-----------+
| NULL      |
+-----------+
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select power(2,null);
Query: select power(2,null)
+------------------+
| power(2.0, null) |
+------------------+
| NULL             |
+------------------+
```

```
Returned 1 row(s) in 0.09s
[hadoop-cs1:21000] > select concat('a',null,'b');
Query: select concat('a',null,'b')
+------------------------+
| concat('a', null, 'b') |
+------------------------+
| NULL                   |
+------------------------+
Returned 1 row(s) in 0.09s
```

聚集函数与其他函数不同，它需要对结果集中所有的项进行运算，因此它需要使用 FROM 子句。

```
[hadoop-cs1:21000] > select count(*) from t1;
Query: select count(*) from t1
+----------+
| count(*) |
+----------+
| 3        |
+----------+
Returned 1 row(s) in 0.31s
[hadoop-cs1:21000] > select x,sum(y) from t1 group by x;
Query: select x,sum(y) from t1 group by x
+---+--------+
| x | sum(y) |
+---+--------+
| 1 | 2      |
| 3 | 4      |
| 5 | 6      |
+---+--------+
Returned 3 row(s) in 0.47s
```

聚集函数不会返回 NULL 值，因为它总是会忽略参与运算的列中的 NULL 值。例如，如果某些行中的列有 NULL 值，那么在对这一列计算 AVG()时，NULL 值的行会被忽略。同样地，如果我们在查询中使用 COUNT(col_name)计算行数，只有那些 col_name 为非 NULL 值的行数才会被计算在内。

4.7.1 数学函数

Impala 支持以下数学函数：

（1）abs(double a)

该函数返回输入参数的绝对值。返回类型为 DOUBLE。

该函数能保证所有返回的值都是正的。它与函数 positive()完全不同，positive()无论输入参数为正值还是负值，返回的值都没有任何变化。

（2）acos(double a)

该函数返回参数的反余弦值。返回类型为 DOUBLE。

（3）asin(double a)

该函数返回参数的反正弦值。返回类型为 DOUBLE。

（4）atan(double a)

该函数返回参数的反正切值。返回类型为 DOUBLE。

（5）bin(bigint a)

该函数返回参数的二进制代码表示，返回值为包含了 0 和 1 的字符串。

（6）ceil(double a), ceiling(double a)

该函数返回大于或等于参数值的最小整数。返回类型为整数。

（7）conv(bigint num, int from_base, int to_base), conv(string num, int from_base, int to_base)

该函数将参数的值转换为指定进制的数值字符串。比如，我们可以将一个 16 进制的数 FCE2 转换为 10 进制数。输入参数可以是整数或者字符串，返回值为字符串类型。如果需要把返回值转换成需要的类型，需要使用 CAST()函数。

（8）cos(double a)

该函数返回参数的余弦值，返回类型为 DOUBLE。

（9）degrees(double a)

该函数将弧度值转换为度数。返回类型为 DOUBLE。

（10）e()

该函数返回自然对数 e，返回类型为 DOUBLE。

（11）exp(double a)

该函数返回自然对数 e 的 a 次方，返回类型为 DOUBLE。

（12）floor(double a)

该函数返回小于或等于输入参数最大整数值。返回类型为 INT。

（13）fmod(double a, double b), fmod(float a, float b)

该函数返回输入参数 a 除以输入参数 b 的余数。返回类型为 FLOAT 或者 DOUBLE，返回类型取决于输入参数的类型。该函数自 Impala 1.1.1 起可使用。

（14）fnv_hash(type v)

该函数返回从输入参数得到的一个 64 位的值，该值可以在应用中很方便的完成 hash 运算。返回类型为 BIGINT。该函数在 Impala 1.2.2 版本起可以使用。

在很多情况下，我们需要在应用程序中基于该函数的返回值来进行将具备某些不同特征的值聚集在一起处理来实现负载均衡或者其他的分布式技术。

因为返回结果是一个 64 位的值，我们可以使用 ABS()绝对值函数和 %取模操作符来将结果限制在某个范围之内。比如，为了产生一个从 0 到 9 的 hash 值，我们可以使用表达式 ABS(FNV_HASH(x)) % 10 来实现。

在不支持 CRC32 算法的系统上，该函数与 Impala 内部使用 hash 算法相同。

该函数是 FNV-1a 的变种。某些不同值组合作为输入参数，有可能产生相同的返回值，所以它不适合用于加密。

示例：

```
[hadoop-cs1:21000] > create table h (x int, s string);
[hadoop-cs1:21000] > insert into h values (0, 'hello'), (1,'world'),
(1234567890,'antidisestablishmentarianism');
[hadoop-cs1:21000] > select x, fnv_hash(x) from h;
+------------+----------------------+
| x | fnv_hash(x) |
+------------+----------------------+
| 0 | -2611523532599129963 |
| 1 | 4307505193096137732 |
| 1234567890 | 3614724209955230832 |
+------------+----------------------+
[hadoop-cs1:21000] > select s, fnv_hash(s) from h;
+------------------------------+---------------------+
| s | fnv_hash(s) |
+------------------------------+---------------------+
| hello | 6414202926103426347 |
| world | 6535280128821139475 |
| antidisestablishmentarianism | -209330013948433970 |
+------------------------------+---------------------+
[hadoop-cs1:21000] > select s, abs(fnv_hash(s)) % 10 from h;
+------------------------------+-------------------------+
| s | abs(fnv_hash(s)) % 10.0 |
+------------------------------+-------------------------+
| hello | 8 |
| world | 6 |
```

```
| antidisestablishmentarianism | 4 |
+-----------------------------+-------------------------+
```

对于较短的输入参数值，返回值的区分度不是很明显:

```
[hadoop-cs1:21000] > create table b (x boolean);
[hadoop-cs1:21000] > insert into b values (true), (true), (false), (false);
[hadoop-cs1:21000] > select x, fnv_hash(x) from b;
+-------+--------------------+
| x | fnv_hash(x) |
+-------+--------------------+
| true | 2062020650953872396 |
| true | 2062020650953872396 |
| false | 2062021750465500607 |
| false | 2062021750465500607 |
+-------+--------------------+
```

（15）greatest(bigint a[, bigint b ...]), greatest(double a[, double b ...]), greatest(string a[,string b ...]), greatest(timestamp a[, timestamp b ...])

该函数返回表达式列表中的最大值。返回值与输入参数中的原始值相同。返回类型与原始的输入参数的类型相同（如果输入参数为 INT 型会被提升为 BIGINT 型，如果输入参数为 FLOAT 型，会被提升为 DOUBLE 型）。可以使用 CAST()函数转换为自己需要的类型。

（16）least(bigint a[, bigint b ...]), least(double a[, double b ...]), least(string a[, stringb ...]), least(timestamp a[, timestamp b ...])

该函数返回表达式列表中的最小数中的原始值相同。返回类型与原始的输入参数的类型相同（如果输入参数为 INT 型会被提升为 BIGINT 型，如果输入参数为 FLOAT 型，会被提升为 DOUBLE 型）。可以使用 CAST()函数转换为自己需要的类型。

（17）hex(bigint a), hex(string a)

该函数返回输入参数的 16 进制表示。返回类型为 STRING。

（18）ln(double a)

该函数返回输入参数的自然对数。返回类型为 DOUBLE。

（19）log(double base, double a)

该函数返回以 base 为底的 a 的对数。返回类型为 DOUBLE。

（20）log10(double a)

该函数返回以 10 为底的 a 的对数。返回类型为 DOUBLE。

（21）log2(double a)

该函数返回以 2 底的 a 的对数。返回类型为 DOUBLE。

（22）negative(int a), negative(double a)

该函数返回输入参数的相反数。返回类型为 INT 或者 DOUBLE，依赖于输入参数的类型。

（23）pi()

该函数返回圆周率常数π。返回类型为 DOUBLE。

（24）pmod(int a, int b), pmod(double a, double b)

该函数返回输入参数 a 除以输入参数 b 的正的余数。返回类型为 INT 或者 DOUBLE，返回类型取决于输入参数的类型。

（25）positive(int a), positive(double a)

该函数返回输入参数的原始值。返回类型为 INT 或者 DOUBLE，依赖于输入参数的类型。

（26）pow(double a, double p), power(double a, double p)

该函数返回 a 的 p 次幂。返回类型为 DOUBLE。

（27）quotient(int a, int b)

该函数返回 a 除以 b 的整数部分。返回类型为 INT。

（28）radians(double a)

该函数返回度数对应的弧度数。返回类型为 DOUBLE。

（29）rand(), rand(int seed)

该函数返回一个介意 0 和 1 之前的随机数。如果以某个特定的数值作为输入参数，它将会返回一个确定的随机数。

在同一个查询中多次调用 rand()，每个 rand()生成的随机数不同。如果需要在一个查询的多个位置使用同一个随机数，可以在每个 rand()调用中使用相同的输入参数作为种子。

（30）rand(unix_timestamp()) from ...， round(double a), round(double a, int d)

该函数依据四舍五入返回整数值。如果有两个输入参数，后一个输入参数代表小数点后保留的位数。如果输入参数为一个，返回值为 BIGINT，如果输入参数有两个，返回值为 DOUBLE。

（31）sign(double a)

该函数为符号函数，如果输入参数为正数返回 1，如果输入参数为负数，返回-1，如果输入参数为 0，返回 0。

返回类型为 INT。

（32）sin(double a)

返回 a 的正弦值，返回类型为 DOUBLE。

（33）sqrt(double a)

返回 a 的平方根。返回类型为 DOUBLE。

（34）tan(double a)

返回 a 的正切值，返回类型为 DOUBLE。

（35）unhex(string a)

返回 16 进制字符串对应的 ASCII 值对应的字符。返回类型为 STRING。

4.7.2　类型转换函数

Impala 支持如下类型转换函数：

cast(expr as type)

该函数一般与其他函数一起使用，用于将其他函数返回的结果转换为指定的数据类型。Impala 对函数输入参数的类型有严格的规定。例如，Impala 不能自动将 DOUBLE 转换为 FLOAT 类型，不能将 BIGINT 类型转换为 INT 类型等。它不会自动实现从高精度到低精度的转换。当输入参数的类型不是 Impala 要求的类型时，需要使用 CAST()做强制转换。

示例：

```
[hadoop-cs1:21000] > select concat('Here are the first ',10,' results.');
Query: select concat('Here are the first ',10,' results.')
ERROR: AnalysisException: No matching function with signature: concat(STRING, TINYINT, STRING).
[hadoop-cs1:21000] > select concat('Here are the first ',cast(10 as string),' results.');
Query: select concat('Here are the first ',cast(10 as string),' results.')
+-------------------------------------------------------------+
| concat('here are the first ', cast(10 as string), ' results.') |
+-------------------------------------------------------------+
| Here are the first 10 results.                           |
+-------------------------------------------------------------+
Returned 1 row(s) in 0.21s
```

4.7.3　时间和日期函数

Impala 的 TIMESTAMP 类型具有日期部分和时间部分，Impala 的日期时间的表示都是基于 TIMESTAMP 进行的。对于像 hour() 或者 minute()这样的抽取单个时间部分的函数，通常返回整数值。而对于像 date_add() 或者 to_date()这样的格式化日期的函数，通常返回字符串值。

Impala 支持如下的日期和时间函数：

（1）adddate(timestamp startdate, int days), adddate(timestamp startdate, bigint days)

该函数针对 startdate 的时间戳增加指定 days 数目的天数。返回类型为 TIMESTAMP。

（2）current_timestamp()

该函数是函数 now()的别名，返回类型为 TIMESTAMP。

（3）date_add(timestamp startdate, int days), date_add(timestamp startdate, interval_expression)

该函数针对 startdate 的时间戳增加指定 days 数目的天数。与函数 adddate()不同的是，第一个参数的类型可以是字符串类型，Impala 会自动将其转换为 TIMESTAMP 类型。第二个参数除了可以以整型方式指定天数之外，还可以使用 INTERVAL 表达式来指定为周、年、小时、秒等其他的时间单元。该函数返回类型为 TIMESTAMP。

（4）date_sub(timestamp startdate, int days), date_sub(timestamp startdate, interval_expression)

该函数针对 startdate 的时间戳减去指定 days 数目的天数。与函数 adddate()不同的是，第一个参数的类型可以是字符串类型，Impala 会自动将其转换为 TIMESTAMP 类型。第二个参数除了可以以整型方式指定天数之外，还可以使用 INTERVAL 表达式来指定为周、年、小时、秒等其他的时间单元。该函数返回类型为 TIMESTAMP。

（5）datediff(string enddate, string startdate)

该函数返回输入参数中得两个字符串表示的日期相差的天数。返回类型为 INT。

（6）day(string date), dayofmonth(string date)

该函数从输入参数字符串表示的日期中返回每个月的第几号。返回类型为 INT。

（7）dayname(string date)

该函数从输入参数字符串表示的日期返回英文表示的星期几。返回值的范围是从“Sunday”到“Saturday”。该函数通常用于在查询中生成报表的语句。除了可以使用这个函数，也可以对 dayofweek()返回的数值使用 CASE 表达式达到同样的效果。返回类型为 STRING。

（8）dayofweek(string date)

该函数从输入参数字符串表示的日期返回是一周的第几天。返回值的范围是从 1（星期六）到 7（星期日）。返回类型为 INT。

（9）dayofyear(timestamp date)

该函数返回输入参数时间戳表示的是一年的第多少天。返回值的范围为 1（1 月 1 号）到 366（12 月 31 号）。返回值为 INT。

（10）days_add(timestamp startdate, int days), days_add(timestamp startdate, bigint days)

该函数表示对输入参数的时间戳 startdate 加上指定的 days 天数。该函数类似于 date_add()，区别在于 days_add()函数的第一个输入参数为 TIMESTAMP 类型。返回类型为 TIMESTAMP。

（11）days_sub(timestamp startdate, int days), days_sub(timestamp startdate, bigint days)

该函数表示对输入参数的时间戳 startdate 减去指定的 days 天数。该函数类似于 date_sub()，区别在于 days_sub()函数的第一个输入参数为 TIMESTAMP 类型。返回类型为 TIMESTAMP。

（12）from_unixtime(bigint unixtime[, string format])

该函数将从 UNIX 纪元时间到指定时间经历的秒数作为输入参数，将其按照指定的格式转换成时间字符串。返回类型为 STRING。

字符串的格式接受所有可以对 TIMESTAMP 指定的时间格式：日期时间，只有日期，只有时间，可以指定的秒的精度等。

在 Impala 1.3.1 或者更高的版本中，我们可以使用不同数量的占位符对日期时间做不同的表示。使用多个 y，d，H，可以在输出中表示为零前导符。对于 M 表示的月份，非零前导符表示为 3，而使用 MM 表示的包含零前导符的形式表示为 03 ，而使用 MMM 则表示为 Mar，Impala 不允许使用更多的 M 来表示月份。一个日期字符串的格式可以表示为"yyyy-MM-dd HH:mm:ss.SSSSSS"，"dd/MM/yyyy HH:mm:ss.SSSSSS"，"MMMdd, yyyy HH.mm.ss (SSSSSS)"等。

示例：

```
[hadoop-cs1:21000] > select from_unixtime(1392394861,"yyyy-MM-dd HH:mm:ss.SSSS");
+-------------------------------------------------------+
| from_unixtime(1392394861, 'yyyy-mm-dd hh:mm:ss.ssss') |
+-------------------------------------------------------+
| 2014-02-14 16:21:01.0000 |
+-------------------------------------------------------+
[hadoop-cs1:21000] > select from_unixtime(1392394861,"yyyy-MM-dd");
+-----------------------------------------+
| from_unixtime(1392394861, 'yyyy-mm-dd') |
+-----------------------------------------+
| 2014-02-14 |
+-----------------------------------------+
[hadoop-cs1:21000] > select from_unixtime(1392394861,"HH:mm:ss.SSSS");
+--------------------------------------------+
| from_unixtime(1392394861, 'hh:mm:ss.ssss') |
+--------------------------------------------+
| 16:21:01.0000 |
+--------------------------------------------+
[hadoop-cs1:21000] > select from_unixtime(1392394861,"HH:mm:ss");
+---------------------------------------+
| from_unixtime(1392394861, 'hh:mm:ss') |
+---------------------------------------+
```

```
| 16:21:01 |
+--------------------------------------+
```

（13）from_utc_timestamp(timestamp, string timezone)

该函数将一个指定的 UTC 时间戳转换为指定时区的本地时间。返回类型为 TIMESTAMP。

（14）hour(string date)

该函数从一个字符串表示的日期中返回小时部分。返回类型为 INT。

（15）hours_add(timestamp date, int hours), hours_add(timestamp date, bigint hours)

该函数返回时间戳加上指定的小时数生成的新时间戳。返回类型为 TIMESTAMP。

（16）hours_sub(timestamp date, int hours), hours_sub(timestamp date, bigint hours)

该函数返回时间戳减去指定的小时数生成的新时间戳。返回类型为 TIMESTAMP。

（17）microseconds_add(timestamp date, int microseconds), microseconds_add(timestamp date, bigint microseconds)

该函数返回时间戳加上指定的毫秒数生成的新时间戳。返回类型为 TIMESTAMP。

（18）microseconds_sub(timestamp date, int microseconds), microseconds_sub(timestamp date, bigint microseconds)

该函数返回时间戳减去指定的毫秒数生成的新时间戳。返回类型为 TIMESTAMP。

（19）minute(string date)

该函数返回字符串所表示的时间的分钟数。返回日期为 INT。

（20）minutes_add(timestamp date, int minutes), minutes_add(timestamp date, bigint minutes)

该函数返回时间戳加上指定的分钟数表示的新时间戳。返回类型为 TIMESTAMP。

（21）minutes_sub(timestamp date, int minutes), minutes_sub(timestamp date, bigint minutes)

该函数返回时间戳减去指定的分钟数表示的新时间戳。返回类型为 TIMESTAMP。

（22）month(string date)

该函数返回字符串代表的日期中得月份。返回类型为 INT。

（23）months_add(timestamp date, int months), months_add(timestamp date, bigint months)

该函数返回时间戳加上指定的月份数表示的新时间戳。返回类型为 TIMESTAMP。

（24）months_sub(timestamp date, int months), months_sub(timestamp date, bigint months)

该函数返回时间戳减去指定的月份数表示的新时间戳。返回类型为 TIMESTAMP。

（25）nanoseconds_add(timestamp date, int nanoseconds), nanoseconds_add(timestamp date, bigint nanoseconds)

该函数返回时间戳加上指定的纳秒数表示的新时间戳。返回类型为 TIMESTAMP。

（26）nanoseconds_sub(timestamp date, int nanoseconds), nanoseconds_sub(timestamp date, bigint nanoseconds)

该函数返回时间戳减去指定的纳秒数表示的新时间戳。返回类型为 TIMESTAMP。

（27）now()

该函数返回当前 UTC 标准日期和时间。返回类型为 TIMESTAMP 。

（28）second(string date)

该函数返回字符串所表示的时间的秒数。返回日期为 INT。

（29）seconds_add(timestamp date, int seconds), seconds_add(timestamp date, bigint seconds)

该函数返回时间戳加上指定的秒数表示的新时间戳。返回类型为 TIMESTAMP。

（30）seconds_sub(timestamp date, int seconds), seconds_sub(timestamp date, bigint seconds)

该函数返回时间戳减去指定的秒数表示的新时间戳。返回类型为 TIMESTAMP。

（31）subdate(timestamp startdate, int days, subdate(timestamp startdate, bigint days,

该函数返回时间戳减去指定的天数生成的新时间戳。返回类型为 TIMESTAMP。该函数与 date_sub()的区别在于本函数的输入参数为时间戳，而不是字符串。

（32）to_date(timestamp)

该函数返回基于某个时间戳的字符串。返回类型为 STRING。

（33）to_utc_timestamp(timestamp, string timezone)

该函数用于将指定时区的本地时间转换成 UTC 标准时间。返回类型为 TIMESTAMP。

（34）unix_timestamp(), unix_timestamp(string date), unix_timestamp(string date, string format)

该函数返回当前时间，或者指定时间与 UNIX 纪元时间之间经历的秒数。返回类型为 BIGINT。

（35）weekofyear(string date)

该函数返回字符串代表的时间为一年中的第多少周。返回值范围为 1 到 53。返回类型为 INT。

（36）weeks_add(timestamp date, int weeks), weeks_add(timestamp date, bigint weeks)

该函数返回时间戳加上指定的周数表示的新时间戳。返回类型为 TIMESTAMP。

（37）weeks_sub(timestamp date, int weeks), weeks_sub(timestamp date, bigint weeks)

该函数返回时间戳减去指定的周数表示的新时间戳。返回类型为 TIMESTAMP。

（38）year(string date)

该函数返回字符串指定的时间的年份。返回类型为 INT。

（39）years_add(timestamp date, int years), years_add(timestamp date, bigint years)

该函数返回时间戳加上指定的年数表示的新时间戳。返回类型为 TIMESTAMP。

（40）years_sub(timestamp date, int years), years_sub(timestamp date, bigint years)

该函数返回时间戳减去指定的年数表示的新时间戳。返回类型为 TIMESTAMP。

4.7.4 条件函数

Impala 支持以下条件函数用来判断是否相等、比较操作符，是否为空等。

（1）CASE a WHEN b THEN c [WHEN d THEN e]... [ELSE f] END

将表达式与多个可能进行比较，如果能匹配到，则返回相应的结果。返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

（2）CASE WHEN a THEN b [WHEN c THEN d]... [ELSE e] END

比较一系列的表达式是否为 TRUE，返回第一个为 TRUE 的表达式对应的值。返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

（3）coalesce(type v1, type v2, ...)

返回输入参数中的第一个非空值，如果所有输入参数均为 NULL，则返回 NULL。

返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

（4）if(boolean condition, type ifTrue, type ifFalseOrNull)

根据一个表达式的结果为 TRUE、FALSE、NULL，来返回相应的结果值。返回类型与输入参数类型相同。

（5）isnull(type a, type ifNotNull)

如果表达式的结果非空，则返回表达式的结果。如果表达式的结果为 NULL，则返回第二个参数值。与 Oracle 中的 nvl()，MYSQL 中的 ifnull()相同。

（6）ifnull(type a, type ifNotNull)

函数 isnull()的别名，与之具有相同的行为。该函数是为了与传统数据库厂商的函数保持兼容。自 Impala 1.3.0 起可以使用。

返回类型与输入参数类型相同。

（7）nullif(expr1,expr2)

这个函数表示如果两个表达式相等，则返回 NULL。如果两个表达式不相等，则返回第一个表达式的值。两个表达式的数据类型必须保持兼容。另外，第一个表达式不能为 NULL，如果它为 NULL，将永远不会与第二个表达式进行匹配。

此函数是以下 CASE 表达式的简写：

```
CASE
WHEN expr1 = expr2 THEN NULL
ELSE expr1
END
```

该函数通常被使用在除法表达式中来防止被除数为零的情况。例如：

```
select 1.0 / nullif(c1,0) as reciprocal from t1;
```

自 Impala 1.3.0 起开始支持该函数。返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

（8）nullifzero(numeric_expr)

如果表达式为 0，则返回 NULL；如果表达式为非零，则返回表达式本身。

自 Impala 1.3.0 起支持该函数。返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

（9）nvl(type a, type ifNotNull)

该函数是 isnull()的别名，自 Impala 1.1 起开始支持。

（10）zeroifnull(numeric_expr)

如果表达式为 NULL，则返回 0；如果表达式为非 NULL，则返回表达式本身。

自 Impala 1.3.0 起支持该函数。返回类型与原始的返回值类型相同。但如果是整型，可能会被提升为 BIGINT 型，如果是 FLOAT 型，可能会被提升为 DOUBLE 型。如果对应的列不允许像 BIGINT 或者 DOUBLE 这样的高精度的类型，需要使用 CAST()进行强制转换。

4.7.5　字符串函数

Impala 支持以下字符串函数：

（1）ascii(string str)

该函数返回首字母的 ASCII 码的值。返回类型为 INT。

（2）char_length(string a), character_length(string a)

该函数返回输入参数中字符串的长度，是 length()的别名。返回类型为 INT。

（3）concat(string a, string b...)

该函数将所有输入参数拼接在一起返回一个单个字符串。返回类型为 STRING。

（4）concat_ws(string sep, string a, string b...)

该函数用于将各输入参数以指定的分隔符连接在一起。返回类型为 STRING。

（5）find_in_set(string str, string strList)

返回一个字符串在一个逗号分隔的字符串中第一次出现的位置。如果各参数都为 NULL，则返回 NULL。如果没有匹配到字符串或者搜索字符串中包含了逗号，则返回 0，返回类型为 INT。

（6）group_concat(string s [, string sep])

该函数用于拼接结果集中的每一行返回一个字符串。如果指定了分隔符，则最终返回的字符串将以指定的分隔符分隔。返回类型为 STRING。concat() 和 concat_ws()用于拼接同一行中的不同列，而 group_concat()用于连接不同行。

（7）initcap(string str)

该函数返回首字母大写的字符串。返回类型为 STRING。

（8）instr(string str, string substr)

该函数返回子串在字符串中第一次出现的位置。返回类型为 INT。

（9）length(string a)

该函数返回字符串的长度。返回类型为 INT。

（10）locate(string substr, string str[, int pos])

该函数返回子串在字符串中第一次出现的位置，与 instr()不同的是这个函数可以指定从某个位置之后开始匹配。返回类型为 INT。

（11）lower(string a), lcase(string a)

该函数将输入字符串改为小写。返回类型为 STRING。

（12）lpad(string str, int len, string pad)

该函数基于第一个输入参数返回一个指定长度的字符串。如果第一个输入参数的长度比指定的长度短，则以第三个输入参数的重复序列进行前导填充。如果第一个输入参数的长度比指定的长度长，则从右侧截断字符串。返回类型为 STRING。

（13）ltrim(string a)

该函数会去掉字符串左侧的空格。返回类型为 STRING。

（14）parse_url(string urlString, string partToExtract [, string keyToExtract])

该函数返回 URL 指定的部分，各部分包括'PROTOCOL'、'HOST'、'PATH'、'REF'、'AUTHORITY'、'FILE'、'USERINFO'，或者'QUERY'，必须使用大写形式。当返回'QUERY'部分时，我们可以指定一个键来返回键值对对应的值。返回类型为 STRING。

这个函数对于传统的 Hadoop 中用来解析 web 日志是非常有效的。我们可以使用这个函数从网络流量数据中抽取单独的字段。

（15）regexp_extract(string subject, string pattern, int index)

该函数返回匹配正则表达式模式的指定分组。分组 0 只的是提取整个字符串。而分组 1、分组 2 等等指的是第一、第二部分。

Impala 支持 Boost 库使用的扩展的 POSIX 正则表达式语法。它与我们在 Perl，Python 中使用的非常类似。它不支持.*? 非贪婪匹配。

以为 impala-shell 解释器使用 \ 作为转义字符，如果需要通过 impala-shell 提交正则表达式需要使用\\作为转义字符。我们可以使用字符类名[[:digit:]]，同时也可以使用\d，当然转义过后是\\d。

示例：分组 0 匹配的是整个字符串。

```
[hadoop-cs1:21000] > select regexp_extract('abcdef123ghi456jkl','.*(\\d+)',0);
+------------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*(\\d+)', 0)  |
+------------------------------------------------------+
| abcdef123ghi456 |
+------------------------------------------------------+
Returned 1 row(s) in 0.11s

而分组1只匹配第一个匹配部分。
[hadoop-cs1:21000] > select regexp_extract('abcdef123ghi456jkl','.*(\\d+)',1);
+------------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*(\\d+)', 1)  |
+------------------------------------------------------+
```

```
| 456 |
+------------------------------------------------------+
Returned 1 row(s) in 0.11s
```

Boost 正则表达式语法不支持.*? 非贪婪匹配。以下示例显示以 .* 开头的字符串将匹配源字符串中最长的满足匹配模式的字符串，本例中就是指以贪婪模式返回最右侧的小写字母串。而已 .*作为开头和结尾的匹配模式，将会返回第一个满足匹配模式的字符串，本例中指的就是返回最左侧的小写字符串，也就是我们说的非贪婪模式。

```
[hadoop-cs1:21000] > select regexp_extract('AbcdBCdefGHI','.*([[:lower:]]+)',1);
+--------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*([[:lower:]]+)', 1) |
+--------------------------------------------------------+
| def |
+--------------------------------------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000]                    >                    select
regexp_extract('AbcdBCdefGHI','.*([[:lower:]]+).*',1);
+----------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*([[:lower:]]+).*', 1) |
+----------------------------------------------------------+
| bcd |
+----------------------------------------------------------+
Returned 1 row(s) in 0.11s
```

（16）regexp_replace(string initial, string pattern, string replacement)

本函数返回满足匹配模式的字符串替换首个输入参数中得子串而得到的字符串。返回类型为 STRING。

示例：替换满足匹配模式的字符串后得到新的字符串。

```
[hadoop-cs1:21000] > select regexp_replace('aaabbbaaa','b+','xyz');
+------------------------------------------+
| regexp_replace('aaabbbaaa', 'b+', 'xyz') |
+------------------------------------------+
| aaaxyzaaa |
+------------------------------------------+
Returned 1 row(s) in 0.11s
```

对包含匹配模式代表的原始字符串处理后，得到新的字符串。

```
[hadoop-cs1:21000] > select regexp_replace('aaabbbaaa','(b+)','<\\1>');
```

```
+-------------------------------------------------+
| regexp_replace('aaabbbaaa', '(b+)', '<\\1>') |
+-------------------------------------------------+
| aaa<bbb>aaa
+-------------------------------------------------+
Returned 1 row(s) in 0.11s
```

去除所有非数字的字符串。

```
[hadoop-cs1:21000] > select regexp_replace('123-456-789','[^[:digit:]]','');
+----------------------------------------------------+
| regexp_replace('123-456-789', '[^[:digit:]]', '') |
+----------------------------------------------------+
| 123456789 |
+----------------------------------------------------+
Returned 1 row(s) in 0.12s
```

（17）repeat(string str, int n)

该函数返回重复了指定次数的字符串。返回类型为 STRING。

（18）reverse(string a)

该函数返回翻转顺序后的字符串。返回类型为 STRING。

（19）rpad(string str, int len, string pad)

该函数基于第一个输入参数返回一个指定长度的字符串。如果第一个输入参数的长度比指定的长度短，则以第三个输入参数的重复序列进行右侧填充。如果第一个输入参数的长度比指定的长度长，则从右侧截断字符串。返回类型为 STRING。

（20）rtrim(string a)

该函数会去掉字符串右侧的空格。返回类型为 STRING。

（21）space(int n)

该函数会返回一个指定数目的空格字符串，可以理解为 repeat(‘ ’,n)的缩写形式。返回类型为 STRING。

（22）strleft(string a, int num_chars)

该函数返回字符串左侧指定数目的字母。可以理解为 substr()的缩写形式。返回类型为 STRING。

（23）strright(string a, int num_chars)

该函数返回字符串右侧指定数目的字母。可以理解为 substr()的缩写形式。返回类型为 STRING。

（24）substr(string a, int start [, int len]), substring(string a, int start [, int len])

该函数返回从指定位置开始返回指定长度的字符串的子串。字符串中得字符的位置从 1 开始编号。返回类型为 STRING。

（25）translate(string input, string from, string to)

该函数返回将第一个输入参数中得子字符串替换为另一子字符串后得到的新字符串。返回类型为 STRING。

（26）trim(string a)

该函数返回去掉字符串首尾的空格后的字符串。效果与 ltrim() 和 rtrim()叠加后的效果。返回类型为 STRING。

（27）upper(string a), ucase(string a)

该函数将字符串中的所有字母转换成大写。返回类型为 STRING。

4.7.6 特殊函数

Impala 支持如下不对任何列进行操作的工具函数：

（1）current_database()

该函数返回连接数据库的会话当前的数据库名称。如果没有切换到其他数据库的情况下，为默认数据库 default。如果使用了 USE 或者 impalad –d 选项，那么显示的就是指定的当前数据库。返回类型为 STRING。

（2）pid()

该函数返回会话连接到的 impalad 进程的进程 ID。我们可以对该进程进行 debug，或者执行 trace，显示进程的参数等等。返回类型为 INT。

（3）user()

该函数返回连接到 imaplad 进程的 Linux 操作系统用户的用户名。通常情况下，使用不包含 FROM 子句的查询进行单次调用，以确认系统的授权设置。一旦我们知道了用户名，我们就可以检查用户所属的组，通过授权策略文件，获得组中的那些角色是可用的信息。返回类型为 STRING。

（4）version()

该函数返回当前我们连接到的 impalad 进程的编译日期及精确的版本号。通常我们使用这个函数来确认自己的版本是否具有特定的功能，另外也可以确认所有的 imapad 节点是否是相同版本。返回类型为 STRING。

4.8 聚集函数

聚集函数是按照一定的规则对数据集进行特定的分类。这些函数将对结果集中的所有结果进行遍历并返回一个结果。

聚集函数总是在运算时忽略对 NULL 值的计算，以避免运算结果出现 NULL 值。比如:如果参与求平均 AVG()运算的某些行包含 NULL 值，如果 NULL 参与该运算，计算出的结果会是 NULL，这显然是不合实际的。另外，COUNT(col_name)的运算也同样是仅对 col_name 中的非空值进行 COUNT。

示例:

```
[hadoop-cs1:21000] > desc student
                  > ;
Query: describe student
+---------+--------+---------+
| name    | type   | comment |
+---------+--------+---------+
| id      | int    |         |
| english | int    |         |
| math    | int    |         |
| mf      | string |         |
+---------+--------+---------+
Returned 4 row(s) in 0.13s
```

本部分将以学生表为例来说明聚集函数的用法。id 为学生的学号，english 为英语成绩，math 为数学成绩，mf 为性别，M 为男生，F 为女生。

4.8.1 AVG

求平均函数用于返回一组数字结果集的平均值。它唯一的参数可以是数字列，返回数字的函数或者表达式。在运算时，列中得 NULL 值将被忽略。如果是一个空表，或者求平均的列的值均为 NULL，AVG 将会返回 NULL。

当查询使用了 GROUP BY 后，则 GROUP BY 的每个聚集将会返回一个值。

该函数的返回类型为 DOUBLE。

示例: 计算男生和女生的平均数学成绩。

```
[hadoop-cs1:21000] > select mf,avg(math) from student group by mf;
Query: select mf,avg(math) from student group by mf
+----+-----------+
```

```
| mf | avg(math) |
+----+-----------+
| M  | 83        |
| F  | 67        |
+----+-----------+
Returned 2 row(s) in 0.53s
```

4.8.2 COUNT

该聚集函数用于返回记录的条数。对于该函数，我们需要明确：

- COUNT(*)会将 NULL 的行计算在内。
- COUNT(col_name)只会计算 col_name 为非 NULL 行的记录。
- 我们可以先进行 DISTINCT 去除某列或者某几列的重复值，在对 DISTINCT 的结果进行 COUNT，可以计算某列或者某几列的非重复值的个数。

当查询使用了 GROUP BY 后，则 GROUP BY 的每个聚集将会返回一个值。

该函数返回类型为 DOUBLE。

示例：计算全班的总人数。

```
[hadoop-cs1:21000] > select count(*) from student;
Query: select count(*) from student
+----------+
| count(*) |
+----------+
| 5        |
+----------+
Returned 1 row(s) in 0.31s
```

计算本班男女生人数：

```
[hadoop-cs1:21000] > select mf,count(*) from student group by mf;
Query: select mf,count(*) from student group by mf
+----+----------+
| mf | count(*) |
+----+----------+
| M  | 3        |
| F  | 2        |
+----+----------+
Returned 2 row(s) in 0.47s
```

4.8.3　GROUP_CONCAT

该函数将结果集中得每行记录拼接成一个字符串。如果指定了分隔符，拼接的时会以指定的分隔符进行分隔。

默认情况下，该函数会对整个结果集拼接成一个字符串。为了包含结果集中的其他列，或者需要产生多个子集拼接的多个字符串，需要对查询使用 GROUP BY 子句。

该函数返回类型为 STRING。

示例：列出本班男生和女生的学号。

```
[hadoop-cs1:21000] > select mf,group_concat( cast(id as string)) from student group by mf;
Query: select mf,group_concat( cast(id as string)) from student group by mf
+----+--------------------------------+
| mf | group_concat(cast(id as string)) |
+----+--------------------------------+
| M  | 1, 4, 3                    |
| F  | 5, 2                       |
+----+--------------------------------+
Returned 2 row(s) in 0.41s
```

4.8.4　MAX

该函数用于返回一组数字结果集中的最大值。它唯一的输入参数可以是数字列，返回数字的函数或者表达式等。在计算时，包含的 NULL 的行将会被忽略。如果表为空表，或者列中所有的值均为 NULL，函数的返回结果就为 NULL。

当查询使用了 GROUP BY 后，则 GROUP BY 的每个聚集将会返回一个值。

该函数的返回类型与输入参数的类型相同。

示例：计算本班男女生数学最高分。

```
[hadoop-cs1:21000] > select mf,max(math) from student group by mf;
Query: select mf,max(math) from student group by mf
+----+-----------+
| mf | max(math) |
+----+-----------+
| M  | 100       |
| F  | 89        |
+----+-----------+
Returned 2 row(s) in 0.55s
```

4.8.5 MIN

该函数用于返回一组数字结果集中的最小值。它唯一的输入参数可以是数字列，返回数字的函数或者表达式等。在计算时，包含 NULL 的行将会被忽略。如果表为空表，或者列中所有的值均为 NULL，函数的返回结果就为 NULL。

当查询使用了 GROUP BY 后，则 GROUP BY 的每个聚集将会返回一个值。

该函数的返回类型与输入参数的类型相同。

示例：计算本班男女生数学最低分。

```
[hadoop-cs1:21000] > select mf,min(math) from student group by mf;
Query: select mf,min(math) from student group by mf
+----+-----------+
| mf | min(math) |
+----+-----------+
| M  | 70        |
| F  | 45        |
+----+-----------+
Returned 2 row(s) in 0.54s
```

4.8.6 NDV

该函数返回与 COUNT(DISTINCT col)的结果近似的数值，即非重复值的个数。它运算的速度要快于 COUNT(DISTINCT col)，因为它使用固定大小的内存进行运算，对于基数非常大的结果求非重复值个数，将大大节省内存资源的使用。

这个函数运算的机制与 Impala 底层处理 COMPUTE STATS 语句计算一个列的非重复值的个数的方式相同。

由于该函数返回的值为一个估计值，尤其在基数非常高或者非常低得时候，它可能不能精确地反映一列有多少个不重复的行。但是如果该函数返回的结果比表中的行数还要多，Impala 在生成执行计划时会自动调整该值。

该函数的返回类型为 BIGINT。

示例：计算本班男女生不重复的英语成绩的个数。

```
[hadoop-cs1:21000] > select mf,english from student;
Query: select mf,english from student
+----+---------+
| mf | english |
+----+---------+
| M  | 99      |
| M  | 99      |
| F  | 84      |
```

```
| F  | 53      |
| M  | 67      |
+----+---------+
Returned 5 row(s) in 0.30s
[hadoop-cs1:21000] > select mf,ndv(english) from student group by mf;
Query: select mf,ndv(english) from student group by mf
+----+--------------+
| mf | ndv(english) |
+----+--------------+
| M  | 2            |
| F  | 2            |
+----+--------------+
Returned 2 row(s) in 0.42s
```

4.8.7　SUM

该函数用于返回一组数字结果集的和。它唯一的输入参数可以是数字列，返回数字的函数或者表达式等。在计算时，包含的 NULL 的行将会被忽略。如果表为空表，或者列中所有的值均为 NULL，函数的返回结果就为 NULL。

当查询使用了 GROUP BY 后，则 GROUP BY 的每个聚集将会返回一个值。

该函数的返回类型为 BIGINT（输入参数为整数）或者 DOUBLE（输入参数为浮点数）。

示例：计算本班男女生数学总分。

```
[hadoop-cs1:21000] > select mf,sum(math) from student group by mf;
Query: select mf,sum(math) from student group by mf
+----+-----------+
| mf | sum(math) |
+----+-----------+
| M  | 249       |
| F  | 134       |
+----+-----------+
Returned 2 row(s) in 0.52s
```

4.9 用户自定义函数 UDF

Impala 的 UDF 允许我们自己编写应用逻辑用于处理列值。比如，UDF 可以引用外部算法库，将多列值合成一列，进行地理空间运算或者其他内嵌的 SQL 操作符或者函数无法完成的运算。

我们可以使用 UDF 来简化生成报表的查询逻辑，也可以在 INSERT … SELECT 拷贝数据时对数据进行灵活的转化处理。这与其他传统数据库中的存储过程或者自定义函数也很类似。

Impala 自 1.2 版本开始支持 UDF：

- 在 Impala 1.1 中，查询中使用 UDF 需要使用 Hive shell，因为 Impala 和 Hive 共享同一个元数据库。我们可以切换到 Hive shell 中执行带有 UDF 的查询，再返回到 Impala 的 shell 中。
- 自 Impala 1.2 开始，Impala 既可以支持基于 C++本地代码编写的高性能 UDF，也可以支持原有的基于 Java 的 Hive UDF。
- UDF 为结果集中的每一行返回一个值，而用户自定义的聚集函数 UDAF，可以为一个结果集返回一个值。目前，Impala 不支持用户自定义的表函数和窗口函数。

4.9.1 UDF 概念

根据我们的使用情况，我们可以编写全新的函数，重用已经写好的基于 Java 的 UDF，或者将基于 Java 的 Hive UDF 移植到基于 C++的 Impala 本地 UDF 上。我们既可以编写为每行记录返回一行的标量函数，也可以编写用于分析处理的复杂聚集函数。下面，我们将讨论使用 UDF 的方方面面。

1. UDF 和 UDAF

在大多数情况下，我们使用 UDF 为每行记录返回一个值。当将其使用在查询中时，针对结果集中的每行将调用一次。

UDAF 则是为一组记录返回一个值。我们使用 UDAF 可以像使用内嵌的 COUNT、MAX()、SUM()、AVG()一样汇总、压缩结果集。

2. 本地 Impala UDF

Impala 支持的 UDF 以 C++书写，另外也支持已经存在的 Hive UDF。Cloudera 推荐使用 C++编写的 UDF，因为使用本地代码编译，性能往往比 Java 编写的 UDF 高数十倍。

3. 使用 Hive UDF

在 Impala 中使用 Hive UDF 需要具备以下条件：

（1）参数和返回值必须是 Impala 支持的数据类型。比如，内嵌类型或者复合类型，Impala 就不支持。

（2）目前，如果 Hive UDF 输入参数和返回类型为 TIMESTAMP 将不被支持。

（3）返回的类型必须是像 Text 或者 IntWritable 一样的“Writable”的类型，而不是像 String 或者 Int 一样的 Java 原生的数据类型。否则，UDF 将返回 NULL。

（4）Hive UDAF 和 UDTF 不被支持。

（5）通常情况下，Java UDF 要比本地 Impala UDF 慢数倍。

为了更好的发挥 Impala 架构的性能优势，强烈建议使用基于 C++的本地 Impala UDF。

理解怎样在 Impala 中重用基于 Java 的 UDF 最好的方式就是把 Hive 内嵌函数和相关的 JAR 包部署在 Impala 上，创建新的不同名称的 UDF。

（1）拷贝包含 Hive 内嵌函数的 JAR 包。例如，JAR 的路径为/usr/lib/hive/lib/hive-exec-0. *. *-cdh*.*.0.jar，此处的*代表版本信息。

（2）使用 jar tf jar_file 查看 JAR 内部的类列表。我们将看到形如 org/apache/hadoop/hive/ql/udf/UDFLower.class 和 org/apache/hadoop/hive/ql/udf/UDFOPNegative.class 这样的类名称。记住我们想修改的函数的名称对应类名。我们需要在 Impala 中为 CREATE FUNCTION 语句提供访问入口，就是把上述的/换成 . 并去掉.class 的后缀，也就是 org.apache.hadoop.hive.ql.udf.UDFLower 和 org.apache.hadoop.hive.ql.udf.UDFOPNegative。

（3）拷贝该文件到 Impala 能够读取的 HDFS 的某个位置。为了简单起见，我们可以将 jar 文件命名为 hive-builtins.jar。

（4）对于每一个我们想通过 Impala 调用的 Java UDF，我们都要执行 CREATE FUNCTION 语句，该语句的 LOCATION 子句指定 JAR 文件在 HDFS 上的全路径，SYMBOL 子句指定完整的类名称。另外，需要注意的是，UDF 需要将其定义存在指定的数据库中，所以如果要调用这个 UDF，我们需要先使用 USE 切换到指定的数据库，或者通过 db_name.function_name 来调用。定义是函数必须使用全新的名称，因为 UDF 不能和内嵌的函数重名。

（5）为我们定义的 UDF 输入正确类型的参数，就可以在查询中进行调用了。传入的参数可以是表列，算术运算或者表达式。另外，我们也可以对函数使用 CAST()，将其转换为我们需要的类型。

示例： 重用 Java UDF lower()。

在如下的 impala-shell 会话中，我们创建了一个 Impala UDF ，名称为 my_lower()。它重用了 Hive 的内嵌函数 lower()。创建完成后，我们就可以在 SQL 中使用该 UDF 了。

```
[hadoop-cs1:21000] > create database udfs;
[hadoop-cs1:21000] > use udfs;
hadoop-cs1:21000] > create function lower(string) returns string location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
ERROR: AnalysisException: Function cannot have the same name as a builtin: lower
[hadoop-cs1:21000] > create function my_lower(string) returns string location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
[hadoop-cs1:21000] > select my_lower('Some String NOT ALREADY LOWERCASE');
+----------------------------------------------------+
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
```

```
| some string not already lowercase |
+---------------------------------------------------+
Returned 1 row(s) in 0.11s
[hadoop-cs1:21000] > create table t2 (s string);
[hadoop-cs1:21000] > insert into t2 values ('lower'),('UPPER'),('Init
cap'),('CamelCase');
Inserted 4 rows in 2.28s
[hadoop-cs1:21000] > select * from t2;
+-----------+
| s |
+-----------+
| lower |
| UPPER |
| Init cap |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[hadoop-cs1:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower |
| upper |
| init cap |
| camelcase |
+------------------+
Returned 4 row(s) in 0.54s
[hadoop-cs1:21000] > select my_lower(concat('ABC ',s,' XYZ')) from t2;
+------------------------------------------+
| udfs.my_lower(concat('abc ', s, ' xyz')) |
+------------------------------------------+
| abc lower xyz |
| abc upper xyz |
| abc init cap xyz |
| abc camelcase xyz |
+------------------------------------------+
Returned 4 row(s) in 0.22s
```

示例：重用 Java UDF negative()。

如下示例演示如何重用 Hive 的内嵌函数 negative()。这个例子告诉我们函数的输入参数的类型必须严格与函数签名匹配。首先，我们创建了一个可以接收整型参数的 UDF。如果给这个 UDF 强制传入一个浮点数，Impala 将会报错。要解决这个问题，要么我们将浮点数强制转换为整型数，或者对函数进行重载，实现一个可传入浮点类型的同名函数。

```
    [hadoop-cs1:21000] > create table t (x int);
    [hadoop-cs1:21000] > insert into t values (1), (2), (4), (100);
    Inserted 4 rows in 1.43s
    [hadoop-cs1:21000] > create function my_neg(bigint) returns bigint location
    '/user/hive/udfs/hive.jar'
symbol='org.apache.hadoop.hive.ql.udf.UDFOPNegative';
    [hadoop-cs1:21000] > select my_neg(4);
    +----------------+
    | udfs.my_neg(4) |
    +----------------+
    | -4 |
    +----------------+
    [hadoop-cs1:21000] > select my_neg(x) from t;
    +----------------+
    | udfs.my_neg(x) |
    +----------------+
    | -2 |
    | -4 |
    | -100 |
    +----------------+
    Returned 3 row(s) in 0.60s
    [hadoop-cs1:21000] > select my_neg(4.0);
    ERROR: AnalysisException: No matching function with signature: udfs.my_neg(FLOAT).
    [hadoop-cs1:21000] > select my_neg(cast(4.0 as int));
    +-------------------------------+
    | udfs.my_neg(cast(4.0 as int)) |
    +-------------------------------+
    | -4 |
    +-------------------------------+
    Returned 1 row(s) in 0.11s
    [hadoop-cs1:21000] > create function my_neg(double) returns double location
    '/user/hive/udfs/hive.jar'
symbol='org.apache.hadoop.hive.ql.udf.UDFOPNegative';
    [hadoop-cs1:21000] > select my_neg(4.0);
```

```
+------------------+
| udfs.my_neg(4.0) |
+------------------+
| -4 |
+------------------+
Returned 1 row(s) in 0.11s
```

4.9.2 安装 UDF 开发包

为了能够在 Impala 上开发 UDF，我们需要安装包含头文件、样例代码、编译配置文件的 impala-udf-devel 包。在 http://archive.cloudera.com/impala/网页上找到与自己操作系统版本匹配的文件夹，进入之后会发现.repo 文件。根据不同的操作系统，我们可以使用 yum、zypper、apt-get 等工具安装名称为 impala-udf-devel 的包。

当我们准备编写自己的 UDF 时，我们可以从 https://github.com/cloudera/impala-udf-samples 网页上下载样例代码和编译脚本。

4.9.3 编写 UDF

在我们进行 UDF 的开发之前，需要确认是否已安装了相应的安装包并下载了示例代码。

在编写 UDF 时：

- 在从 SQL 向 UDF 中传入参数时，两者的数据类型是否相同。比如，在 UDF 中，我们可能更关注传入的整型参数是 TINYINT、SMALLINT、INT 还是 BIGINT。
- 使用面向函数编程思想：选择合适的输入参数，让每个函数完成一个独立的功能。

1. UDF 编程例子

为了了解成员变量，预定义 UDF 数据类型等信息，我们可以查看头文件 /usr/include/impala_udf/udf.h。

```
// This is the only Impala header required to develop UDFs and UDAs. This header
// contains the types that need to be used and the FunctionContext object. The context
// object serves as the interface object between the UDF/UDA and the impala process
```

对于编写一个标量 UDF 需要的基本的声明，我们可以查看 udf-sample.h。这个文件中定义了一个名称为 AddUdf()的示例函数。

```
#ifndef IMPALA_UDF_SAMPLE_UDF_H
#define IMPALA_UDF_SAMPLE_UDF_H
#include <impala_udf/udf.h>
using namespace impala_udf;
```

```
IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const IntVal& arg2);
#endif
```

示例函数 AddUdf()的源代码对应的源文件名称为 udf-sample.cc

```
#include "udf-sample.h"
// In this sample we are declaring a UDF that adds two ints and returns an int.
IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const IntVal& arg2)
{
if (arg1.is_null || arg2.is_null) return IntVal::null();
return IntVal(arg1.val + arg2.val);
}
// Multiple UDFs can be defined in the same file
```

2. 函数的输入参数和返回值的数据类型

每个 UDF 的输入参数和返回值都必须能够和表列对应的 SQL 数据类型相匹配。

每个数据类型无论在 C++还是 Java 头文件中都有一个结构体定义。在结构体中定义成员字段和预定义的比较操作符和构造函数：

（1）is_null 表示值是否为空。当返回值非空时，val 处理过参数值就是其原始值。

（2）每个结构体都可以定义一个叫做 null()的成员方法。这个方法可以根据 is_null 标志位构造一个结构体的实例。

（3）像< 、> 、= 、BETWEEN 、ORDER BY 等比较操作符或者语句都可以在 UDF 中正常工作。比如，无须在 UDF 中特别的声明或处理，Impala 可以识别并计算像 BETWEEN 1 AND udf_returning_int(col1) 或者 ORDER BY udf_returning_string(col2) 这样的语句。

为了更方便的对结构体进行比较，每个结构体定义了操作符== 和!=用于比较同类型的结构体。这种比较与 SQL 本身无关，指的是 C++语法的比较。比如，如果两个结构体中都包含成员变量 is_null，而且该标志位的值都是 NULL，则也认为这两个结构体的比较是相等的。这与 SQL 中 NULL 的比较的结果完全不同。

（4）每种结构体都有一个或者多个构造方法来定义结构体的实例。

（5）目前不能使用内嵌类型或者复合类型作为 UDF 的输入参数和返回值。

（6）我们可以使用不同的函数签名重载一个 UDF。

在 C++中数据类型的定义对应如下：

- IntVal 对应 INT。
- BigIntVal 对应 BIGINT。即使我们不需要 BIGINT 那么大的范围，使用 BigIntVal 作为函数参数可以很方便地调用不同类型的整型数据或者表达式。因为 Impala 可以自动地从范围比较小的类型向范围比较大的类型转换，却不能隐式的将一个范围比较大的类型转换为范围比较小的类型。
- SmallIntVal 对应 SMALLINT。

- TinyIntVal 对应 TINYINT。
- StringVal 对应 STRING。它有一个 len 字段表示字符串的长度，prt 字段指向 String 数据的地址。他拥有一个构造函数可以基于一个 C 格式的字符串，也可以基于一个地址指针和长度创建一个新的 StringVal 结构体。这些新的结构体仍然是指向原来的数据，而不是分配一个新的数据缓冲区。它也拥有一个带有 FunctionContext 结构体和长度指针的构造函数，这允许不必为新的字符串数据分配新的空间即可从 UDF 返回字符串数据。
- BooleanVal 对应 BOOLEAN。
- FloatVal 对应 FLOAT。
- DoubleVal 对应 DOUBLE。
- TiemstampVal 对应 TIMESTAMP。它拥有一个 date 字段，使用 32 位整数记录自 UNIX 纪元时间经历的天数。它还有一个 time_of_day 字段，使用 64 位整数记录到当前时间经历的纳秒数。

3. 可变长参数列表

UDF 通常都使用固定数目的参数，每一个参数都在函数签名中显示声明。另外，UDF 还可以接受相同类型的可选参数。比如，我们可以将两个，三个，四个，甚至更多字符串合并。或者我们可以比较两个，三个，四个，甚至多个数字。

为了接受可变长的参数列表，代码中函数的签名应该形如：

```
StringVal Concat(FunctionContext* context, const StringVal& separator,
int num_var_args, const StringVal* args);
```

在 SQL 查询中调用时，我们必须为可变长的参数部分提供至少一个参数。

当 Impala 调用该函数时，它首先会初始化必选参数，然后通过可选参数的数目和指向首个可选参数的指针提取所有的可选参数。

4. 捕获 NULL 值

从正确性、性能和可靠性的角度考虑，UDF 可以正常地处理输入参数中的 NULL 值是非常重要的。例如，当给函数传入一个 NULL 值时，它会返回一个 NULL 值。在一个聚集函数中，它必须能够将具体数值和 NULL 一起处理，这种处理可能是返回 NULL(比如 CONCAT())，忽略 NULL 值（比如 AVG()），或者将其作为 0 或者”空字符串对待。

像 IntVal 或者 StringVal 这样的每一个参数类型，都有一个 is_null 布尔成员。为每一个函数的输入参数测试这个标志位，如果设置了这个标志位，就不会指向 val 字段。如果输入参数为 NULL，val 字段就不会被定义，我们的函数可能因此进去无线循环或者产生不正确的结果。因为，我们必须捕获处理输入参数为 NULL 的情况。

如果我们输入参数为 NULL，返回 NULL 值。在另外一些情况下，比如我们搜索一个字符串但是没有找到，我们也可以使用 isnull()成员函数构造一个指定返回类型的空的实例。

5. UDF 的内存分配

默认情况下，当函数退出时，UDF 释放内存，这个过程可能在查询结束之前就开始进行了。

函数的输入参数与函数本身具有相同的生命周期，因为我们可以在包含返回值的表达式中引用它们。如果我们使用临时变量构造一个全新的字符串，我们需要使用一个 FunctionContext*初始化参数和一个长度的 StringVal()的构造器，然后将数据拷贝到新的内存缓冲区中。

6. UDF 线程安全工作区

提升 UDF 性能的一种方式就是在 CREATE FUNCTION 语句中指定 PREPARE_FN 和 CLOSE_FN。PREPARE 函数在内存中创建一个线程安全的数据结构，我们可以把它作为工作区使用。CLOSE 函数则负责释放这块内存。后续每次在同一线程内的调用都可以访问相同的内存区域。在 UDF 使用多个线程时，在同一主机上可能有多个这样的内存区域。

在这个内存区域中，我们可以预定义查找表，也可以记录像 STRING 或者 TIMESTAMP 这样数据类型的复杂操作的结果。我们如果能够保存之前的运算结果，就可以避免每次对需要的结果重新计算，大大提高效率。例如，对包含很多重复值的某列执行正则表达式匹配或者日期函数操作，如果没有工作区，我们需要每次都对相同的值执行重复的运算。而在工作区中，我们可以通过缓存最近计算的记录或者对已经计算过的值建立一张哈希表，对于那些重复值就不再需要重复的运算，直接在工作区中进行查找即可获得需要的结果。

每个这样的函数必须有以下签名：

```
void function_name(impala_udf::FunctionContext*,
impala_udf::FunctionContext::FunctionScope)
```

目前，只有 THREAD_SCOPE 部分实现了，而 FRAGMENT_SCOPE 部分没有实现。我们可以从 udf.H 中了解更多细节。

7. UDF 错误捕获

为了捕获 UDF 错误，我们需要将初始化参数 FunctionContext*的成员传递给我们的函数。

UDF 可以记录警告信息，而对于那些不重要的、可恢复的警告信息不会导致 Impala 终止查询。函数的签名形如：

```
bool AddWarning(const char* warning_msg);
```

对于那些可能导致查询终止的严重问题，UDF 可以设置一个错误标志位来阻止查询继续进行。函数的签名形如：

```
void SetError(const char* error_msg);
```

4.9.4　编写 UDAF

UDAF 需要为后续的调用维护一个状态值，因为它是对一个结果集的一组调用返回的累计

值。因此，UDAF 必须包含以下功能：

- 一个初始化函数用于完成计数器清零，创建内存缓冲区和其他查询执行前的初始化操作。
- 一个更新函数用于处理查询结果集的每行记录，并对每个节点的中间结果进行累计。
- 一个合并函数用于将不同节点的中间结果进行合并。
- 一个结束函数用于完成最终转换或者结果合并。

按照 SQL 的语法，我们可以使用 CREATE AGGREATE FUNCTION 语句创建 UDAF。我们可以使用 INIT_FN、UPDATE_FN、MERGE_FN、FINALIZE_FN 来调用 C++底层函数。

方便起见，我们可以使用对底层函数的命名约定来命名函数，这样 Impala 会自动匹配到这些函数。比如，对于 UPDATE_FN，我们可以使用 update 或者 Update 来命名。

uda-sample.h

```
#ifndef IMPALA_UDF_SAMPLE_UDA_H
#define IMPALA_UDF_SAMPLE_UDA_H
#include <impala_udf/udf.h>
using namespace impala_udf;
void CountInit(FunctionContext* context, BigIntVal* val);
void CountUpdate(FunctionContext* context, const AnyVal& input, BigIntVal* val);
void CountMerge(FunctionContext* context, const BigIntVal& src, BigIntVal* dst);
BigIntVal CountFinalize(FunctionContext* context, const BigIntVal& val);
void AvgInit(FunctionContext* context, BufferVal* val);
void AvgUpdate(FunctionContext* context, const DoubleVal& input, BufferVal* val);
void AvgMerge(FunctionContext* context, const BufferVal& src, BufferVal* dst);
DoubleVal AvgFinalize(FunctionContext* context, const BufferVal& val);
void StringConcatInit(FunctionContext* context, StringVal* val);
void StringConcatUpdate(FunctionContext* context, const StringVal& arg1,
const StringVal& arg2, StringVal* val);
void StringConcatMerge(FunctionContext* context, const StringVal& src, StringVal*
dst);
StringVal StringConcatFinalize(FunctionContext* context, const StringVal& val);
#endif
```

uda-sample.cc:

```
#include "uda-sample.h"
#include <assert.h>
using namespace impala_udf;
// ---------------------------------------------------------------------------
// COUNT 示例
```

```
// ---------------------------------------------------------------------------
void CountInit(FunctionContext* context, BigIntVal* val) {
val->is_null = false;
val->val = 0;
}
void CountUpdate(FunctionContext* context, const AnyVal& input, BigIntVal* val)
{
if (input.is_null) return;
++val->val;
}
void CountMerge(FunctionContext* context, const BigIntVal& src, BigIntVal* dst)
{
dst->val += src.val;
}
BigIntVal CountFinalize(FunctionContext* context, const BigIntVal& val) {
return val;
}
// ---------------------------------------------------------------------------
// AVG 示例
// ---------------------------------------------------------------------------
struct AvgStruct {
double sum;
int64_t count;
};
void AvgInit(FunctionContext* context, BufferVal* val) {
assert(sizeof(AvgStruct) == 16);
memset(*val, 0, sizeof(AvgStruct));
}
void AvgUpdate(FunctionContext* context, const DoubleVal& input, BufferVal* val)
{
if (input.is_null) return;
AvgStruct* avg = reinterpret_cast<AvgStruct*>(*val);
avg->sum += input.val;
++avg->count;
}
void AvgMerge(FunctionContext* context, const BufferVal& src, BufferVal* dst) {
if (src == NULL) return;
const AvgStruct* src_struct = reinterpret_cast<const AvgStruct*>(src);
AvgStruct* dst_struct = reinterpret_cast<AvgStruct*>(*dst);
```

```
dst_struct->sum += src_struct->sum;
dst_struct->count += src_struct->count;
}
DoubleVal AvgFinalize(FunctionContext* context, const BufferVal& val) {
if (val == NULL) return DoubleVal::null();
AvgStruct* val_struct = reinterpret_cast<AvgStruct*>(val);
return DoubleVal(val_struct->sum / val_struct->count);
}
// ---------------------------------------------------------------------------
// 类似 GROUP_CONCAT 示例
// ---------------------------------------------------------------------------
void StringConcatInit(FunctionContext* context, StringVal* val) {
val->is_null = true;
}
void StringConcatUpdate(FunctionContext* context, const StringVal& arg1,
const StringVal& arg2, StringVal* val) {
if (val->is_null) {
val->is_null = false;
*val = StringVal(context, arg1.len);
memcpy(val->ptr, arg1.ptr, arg1.len);
} else {
int new_len = val->len + arg1.len + arg2.len;
StringVal new_val(context, new_len);
memcpy(new_val.ptr, val->ptr, val->len);
memcpy(new_val.ptr + val->len, arg2.ptr, arg2.len);
memcpy(new_val.ptr + val->len + arg2.len, arg1.ptr, arg1.len);
*val = new_val;
}
}
void StringConcatMerge(FunctionContext* context, const StringVal& src, StringVal*
dst)
{
if (src.is_null) return;
StringConcatUpdate(context, src, ",", dst);
}
StringVal StringConcatFinalize(FunctionContext* context, const StringVal& val) {
return val;
}
```

4.9.5　编译和部署 UDF

本部分介绍编译 Impala UDF 的 C++代码，部署代码库，并将其使用在 Impala 的查询中。

Impala 自带了一个 UDF 示例编译环境，我们可以在这个环境上编译，测试自己的 UDF。在示例编译环境中，我们首先执行 cmake 命令，它将读取 CmakeLists.txt 文件，并在指定的路径生成 Makefile 文件。然后，我们执行 make 命令，它将依据 Makefile 中定义的规则来执行真正的编译过程。

Impala 从 HDFS 上来加载共享库。在编译了包含一个或者多个 UDF 的共享库之后，我们可以使用 hdfs dfs 或者 hadoop fs 命令将二进制文件拷贝到 Impala 具有可读权限的 HDFS 的某个位置上。

最后的步骤就是在 impala-shell 解释器中执行 CREATE FUNCTION 语句。

在我们更新了 UDF 代码或者重新部署了新版本的共享库之后，为了让函数识别到最新的代码，我们需要使用 DROP FUNCTION 和 CREATE FUNCTION 重建该函数。

编译环境需要的包及安装命令如下：

- sudo yum install gcc-c++ cmake boost-devel
- sudo yum install impala-udf-devel

然后，我们解压示例代码 udf_samples.tar.gz，将其作为一个模板生成编译环境。

为了编译原始的示例代码：

```
cmake
make
```

示例代码将对以下文件进行检查，确认：

（1）udf-sample.h:声明标量 UDF 签名的头文件。

（2）udf-sample.cc:一个简单的将两个整数相加的示例 UDF 源代码。因为 Impala 可以对同一个共享库引用多个函数，所以我们可以将其他的 UDF 添加到这个文件中，同时将它们的函数签名添加到相关的头文件中。

（3）udf-sample-test.cc:示例 UDF 的单元测试用例。

（4）uda-sample.h:声明 UDAF 的签名的头文件。这些函数是 COUNT、AVG 和 STRINGCONCAT。因为聚集函数需要使用更详细的代码捕获不同的处理过程，所以其中包含了像 CountInit、AvgUpdate、StringConcatFinalize 等底层函数。

（5）uda-sample.cc:UDAF 示例源代码，用来演示在底层函数被调用的不同阶段如何进行状态转换。

UDAF 模拟 COUNT 函数跟踪一个递增的数字；在每个 Impala 节点合并中间结果；并将最终合并的结果返回。

UDAF 模拟 AVG 函数跟踪两个数字，一个是处理的记录的行数，一个是该列所有值的和；

这些值像 COUNT 处理过程一样更新合并中间结果；最终返回两个数字相除的结果，也就是平均值。

UDAF 将不同行的值拼接成一个以逗号分隔的字符串，这个示例演示了在对多行处理时，如何对长度递增的字符串进行存储管理。

（6）uda-sample-test.cc:示例 UDAF 的单元测试用例。

4.9.6 UDF 性能

通常情况下，UDF 要对表中的每一行进行处理。如果 ETL 或者 ELT 的数据量有十亿条，那么 UDF 也会处理十亿次。ETL 或者 ELT 整体的速度将会成为影响 UDF 效率的关键因素。对于一个对海量数据集反复调用的 UDF，我们对函数体任何细小的优化，都可能带来巨大的性能提升。

4.9.7 创建和使用 UDF 示例

本部分将演示如何创建和使用不同类型的 UDF。

C++ UDF 示例：HasVowels，CountVowels，StripVowels。

该示例演示的是三个对字符串进行操作并返回不同数据类型返回值的 UDF。在 C++代码中，函数分别为 HasVowels()（检查字符串中是否包含元音字母），CountVowels()（返回字符串中元音字母的个数），StripVowels()（返回一个去掉了所有元音字母的字符串）。

首先，我们要将这些函数的签名添加到 udf-sample.h 的头文件中：

```
BooleanVal HasVowels(FunctionContext* context, const StringVal& input);
IntVal CountVowels(FunctionContext* context, const StringVal& arg1);
StringVal StripVowels(FunctionContext* context, const StringVal& arg1);
```

然后，我们将函数体添加到 udf-sample.cc 中：

```
BooleanVal HasVowels(FunctionContext* context, const StringVal& input)
{
if (input.is_null) return BooleanVal::null();
int index;
uint8_t *ptr;
for (ptr = input.ptr, index = 0; index <= input.len; index++, ptr++)
{
uint8_t c = tolower(*ptr);
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
{
return BooleanVal(true);
```

```
}
}
return BooleanVal(false);
}
IntVal CountVowels(FunctionContext* context, const StringVal& arg1)
{
if (arg1.is_null) return IntVal::null();
int count;
int index;
uint8_t *ptr;
for (ptr = arg1.ptr, count = 0, index = 0; index <= arg1.len; index++, ptr++)
{
uint8_t c = tolower(*ptr);
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
{
count++;
}
}
return IntVal(count);
}
StringVal StripVowels(FunctionContext* context, const StringVal& arg1)
{
if (arg1.is_null) return StringVal::null();
int index;
std::string original((const char *)arg1.ptr,arg1.len);
std::string shorter("");
for (index = 0; index < original.length(); index++)
{
uint8_t c = original[index];
uint8_t l = tolower(c);
if (l == 'a' || l == 'e' || l == 'i' || l == 'o' || l == 'u')
{
;
}
else
{
shorter.append(1, (char)c);
}
}
```

```
StringVal result(context, shorter.size()); // Only the version of the ctor that
takes a context object allocates new memory
memcpy(result.ptr, shorter.c_str(), shorter.size());
return result;
}
```

我们编译共享库 libudfsample.so，并将库文件上传到 Impala 有读权限的 HDFS 上：

```
$ make
[ 0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
[ 33%] Built target udasample
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
Scanning dependencies of target udfsample
[ 83%] Building CXX object CMakeFiles/udfsample.dir/udf-sample.o
Linking CXX shared library udf_samples/libudfsample.so
[ 83%] Built target udfsample
Linking CXX executable udf_samples/udf-sample-test
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudfsample.so /user/hive/udfs/libudfsample.so
```

最后，我们在 impala-shell 解释器中执行 CREATE FUNCTION 语句，创建该 UDF：

```
[hadoop-cs1:21000] > create database udf_testing;
[hadoop-cs1:21000] > use udf_testing;
[hadoop-cs1:21000] > create function has_vowels (string) returns boolean location
'/user/hive/udfs/libudfsample.so' symbol='HasVowels';
[hadoop-cs1:21000] > select has_vowels('abc');
+------------------------+
| udfs.has_vowels('abc') |
+------------------------+
| true |
+------------------------+
Returned 1 row(s) in 0.13s
[hadoop-cs1:21000] > select has_vowels('zxcvbnm');
+----------------------------+
| udfs.has_vowels('zxcvbnm') |
+----------------------------+
| false |
```

```
+----------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > select has_vowels(null);
+-----------------------+
| udfs.has_vowels(null) |
+-----------------------+
| NULL |
+-----------------------+
Returned 1 row(s) in 0.11s
[hadoop-cs1:21000] > select s, has_vowels(s) from t2;
+-----------+--------------------+
| s | udfs.has_vowels(s) |
+-----------+--------------------+
| lower | true |
| UPPER | true |
| Init cap | true |
| CamelCase | true |
+-----------+--------------------+
Returned 4 row(s) in 0.24s
[hadoop-cs1:21000] > create function count_vowels (string) returns int location
'/user/hive/udfs/libudfsample.so' symbol='CountVowels';
[hadoop-cs1:21000] > select count_vowels('cat in the hat');
+-------------------------------------+
| udfs.count_vowels('cat in the hat') |
+-------------------------------------+
| 4 |
+-------------------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > select s, count_vowels(s) from t2;
+-----------+----------------------+
| s | udfs.count_vowels(s) |
+-----------+----------------------+
| lower | 2 |
| UPPER | 2 |
| Init cap | 3 |
| CamelCase | 4 |
+-----------+----------------------+
Returned 4 row(s) in 0.23s
[hadoop-cs1:21000] > select count_vowels(null);
```

```
+-------------------------+
| udfs.count_vowels(null) |
+-------------------------+
| NULL |
+-------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > create function strip_vowels (string) returns string location
'/user/hive/udfs/libudfsample.so' symbol='StripVowels';
[hadoop-cs1:21000] > select strip_vowels('abcdefg');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| bcdfg |
+------------------------------+
Returned 1 row(s) in 0.11s
[hadoop-cs1:21000] > select strip_vowels('ABCDEFG');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| BCDFG |
+------------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > select strip_vowels(null);
+-------------------------+
| udfs.strip_vowels(null) |
+-------------------------+
| NULL |
+-------------------------+
Returned 1 row(s) in 0.16s
[hadoop-cs1:21000] > select s, strip_vowels(s) from t2;
+-----------+----------------------+
| s | udfs.strip_vowels(s) |
+-----------+----------------------+
| lower | lwr |
| UPPER | PPR |
| Init cap | nt cp |
| CamelCase | CmlCs |
+-----------+----------------------+
Returned 4 row(s) in 0.24s
```

C++ UDAF 示例：SumOfSquares。

该示例演示了一个用户自定义的对输入值求平方和的聚集函数。

UDAF 的编码与标量 UDF 的编码有些不同。因为 UDAF 将其划分为不同的处理过程，而每个过程使用不同的函数实现。像其中的 UPDATE 和 MERGE 阶段是比较简单的，主要是负责读取输入值，并将中间累计结果进行合并。

与之前的标量 UDF 类似，我们需要将函数的签名添加到头文件中（本示例中是将其添加到 uda-sample.h 中）。因为 UDAF 都涉及数学运算，所以我们需要实现两个函数的版本，一个是接收整型数据的，一个是接收浮点型数据的。

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val);
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val);
void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal& input,
BigIntVal*
val);
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal& input,
DoubleVal*
val);
void SumOfSquaresMerge(FunctionContext* context, const BigIntVal& src, BigIntVal*
dst);
void SumOfSquaresMerge(FunctionContext* context, const DoubleVal& src, DoubleVal*
dst);
BigIntVal SumOfSquaresFinalize(FunctionContext* context, const BigIntVal& val);
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const DoubleVal& val);
```

我们将函数体添加到 C++源文件中（本例中为 uda-sample.cc）：

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val) {
val->is_null = false;
val->val = 0;
}
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val) {
val->is_null = false;
val->val = 0.0;
}
void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal& input,
BigIntVal*
val) {
if (input.is_null) return;
val->val += input.val * input.val;
```

```
}
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal& input,
DoubleVal*
val) {
if (input.is_null) return;
val->val += input.val * input.val;
}
void SumOfSquaresMerge(FunctionContext* context, const BigIntVal& src, BigIntVal*
dst)
{
dst->val += src.val;
}
void SumOfSquaresMerge(FunctionContext* context, const DoubleVal& src, DoubleVal*
dst)
{
dst->val += src.val;
}
BigIntVal SumOfSquaresFinalize(FunctionContext* context, const BigIntVal& val) {
return val;
}
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const DoubleVal& val) {
return val;
}
```

然后，我们编译共享库，并将其上传到 HDFS 上：

```
$ make
[ 0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
Scanning dependencies of target udasample
[ 33%] Building CXX object CMakeFiles/udasample.dir/uda-sample.o
Linking CXX shared library udf_samples/libudasample.so
[ 33%] Built target udasample
Scanning dependencies of target uda-sample-test
[ 50%] Building CXX object CMakeFiles/uda-sample-test.dir/uda-sample-test.o
Linking CXX executable udf_samples/uda-sample-test
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
[ 83%] Built target udfsample
```

```
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudasample.so /user/hive/udfs/libudasample.so
```

之后，我们在 impala-shell 解释器中执行 CREATE AGGREAGTE FUNCTION 语句，完成 UDAF 的创建：

```
[hadoop-cs1:21000] > use udf_testing;
[hadoop-cs1:21000] > create table sos (x bigint, y double);
[hadoop-cs1:21000] > insert into sos values (1, 1.1), (2, 2.2), (3, 3.3), (4, 4.4);
Inserted 4 rows in 1.10s
[hadoop-cs1:21000] > create aggregate function sum_of_squares(bigint) returns
bigint
> location '/user/hive/udfs/libudasample.so'
> init_fn='SumOfSquaresInit'
> update_fn='SumOfSquaresUpdate'
> merge_fn='SumOfSquaresMerge'
> finalize_fn='SumOfSquaresFinalize';
[hadoop-cs1:21000] > -- Compute the same value using literals or the UDA;
[hadoop-cs1:21000] > select 1*1 + 2*2 + 3*3 + 4*4;
+-------------------------------+
| 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 |
+-------------------------------+
| 30 |
+-------------------------------+
Returned 1 row(s) in 0.12s
[hadoop-cs1:21000] > select sum_of_squares(x) from sos;
+------------------------+
| udfs.sum_of_squares(x) |
+------------------------+
| 30 |
+------------------------+
Returned 1 row(s) in 0.35s
```

到目前为止，我们的 UDAF 只能接收整型数据作为输入参数，为了可以将浮点数作为输入参数，我们继续执行 CREATE AGGREGATE FUNCTION 语句：

```
[hadoop-cs1:21000] > select sum_of_squares(y) from sos;
ERROR: AnalysisException: No matching function with signature:
udfs.sum_of_squares(DOUBLE).
[hadoop-cs1:21000] > create aggregate function sum_of_squares(double) returns
```

```
double
    > location '/user/hive/udfs/libudasample.so'
    > init_fn='SumOfSquaresInit'
    > update_fn='SumOfSquaresUpdate'
    > merge_fn='SumOfSquaresMerge'
    > finalize_fn='SumOfSquaresFinalize';
    [hadoop-cs1:21000] > -- Compute the same value using literals or the UDA;
    [hadoop-cs1:21000] > select 1.1*1.1 + 2.2*2.2 + 3.3*3.3 + 4.4*4.4;
    +-----------------------------------------------+
    | 1.1 * 1.1 + 2.2 * 2.2 + 3.3 * 3.3 + 4.4 * 4.4 |
    +-----------------------------------------------+
    | 36.3 |
    +-----------------------------------------------+
    Returned 1 row(s) in 0.12s
    [hadoop-cs1:21000] > select sum_of_squares(y) from sos;
    +-----------------------+
    | udfs.sum_of_squares(y) |
    +-----------------------+
    | 36.3 |
    +-----------------------+
    Returned 1 row(s) in 0.35s
```

通常情况下，我们会将 UDAF 与 GROUP BY 一起使用，用于对分组后的每一个结果集进行运算。如下示例中，我们使用 0 标识偶数行，使用 1 标识奇数行。使用 GROUP BY 对数据按照奇偶数进行分组，并对每组数据分别求平方和。

```
    [hadoop-cs1:21000] > insert overwrite sos values (1, 1), (2, 0), (3, 1), (4, 0);
    Inserted 4 rows in 1.24s
    [hadoop-cs1:21000] > -- Compute 1 squared + 3 squared, and 2 squared + 4 squared;
    [hadoop-cs1:21000] > select y, sum_of_squares(x) from sos group by y;
    +---+-----------------------+
    | y | udfs.sum_of_squares(x) |
    +---+-----------------------+
    | 1 | 10 |
    | 0 | 20 |
    +---+-----------------------+
    Returned 2 row(s) in 0.43s
```

4.9.8　UDF 安全

当 Impala 开启授权机制特性后：

- 为了在查询中使用 UDF，我们必须对查询中的所有的数据库及相关的表有读取权限。
- 为了尽可能地避免 UDF 由于编码错误导致的无限循环或者内存消耗过大等问题，只有管理员才可以创建 UDF。也就是说，只有拥有服务器的 ALL 权限才可以执行 CREATE FUNCTION 语句。

4.9.9　Impala UDF 的限制

在当前版本，Impala 对 UDF 有以下限制：

- Impala 不支持在 Hive UDF 中可以使用的内嵌类型、复合类型以及其他 Impala 表不支持的类型。
- 所有的 Impala UDF 必须是确定的，也就是说每次我对 UDF 输入相同的输入参数，返回的值肯定是相同的。例如，Impala 不支持在 UDF 中调用 rand()，因为这样会导致相同的输入参数产生不同的返回值。另外，它也不支持从磁盘、网络等外部数据源获取数据。
- Impala UDF 不能产生其他的线程或者进程。
- 当 catalogd 进程重启之后，所有的 UDF 必须重新加载。
- Impala 当前不支持 UDTF（用户自定义的表函数）。

4.10　Impala SQL &Hive QL

当前 Impala 版本不支持但在 Hive 中可以使用的如下功能：

- 像 map，array，struct 等非标量数据类型。
- 像 TRANSFORM，自定义文件格式，自定义 SerDes 等扩展特性。
- XML 和 JSON 函数。
- Hive 中的某些聚集函数：variance、var_pop、var_samp、stddev_pop、stddev_samp、covar_pop、covar_samp、corr、percentile、percentile_approx、histogram_numeric、collect_set。
- 采样。
- Hive 中用于行列转换的横向视图。
- 单个查询中使用多个 DISTINCT。

Impala 当前还不支持如下 HiveQL 语句：

```
ANALYZE TABLE (Impala 中使用 COMPUTE STATS)
DESCRIBE COLUMN
DESCRIBE DATABASE
EXPORT TABLE
IMPORT TABLE
SHOW PARTITIONS
SHOW TABLE EXTENDED
SHOW INDEXES
SHOW COLUMNS
```

在一些情况下 HiveQL 与 Impala 的 SQL 语句非常相似，却有不同的含义：

（1）Impala 为 Hint 使用不同的语法和命名。

（2）Impala 在执行 SORT BY、DISTRIBUTE BY、CLUSTER BY 时不使用 MapReduce。

（3）Impala 查询中可以不需要 FROM 子句。

（4）Impala 支持有限的几种隐式转换。这样做是为了避免由于隐式转换而导致的不可预知的结果。

Impala 不支持在字符串和数字、布尔类型之间的隐式转换。如果要进行这些转换，必须使用 CAST()强制进行。

Impala 只支持从范围小精度低的数字向范围大精度高的数字进行隐式转换。例如，Impala 可以对 SMALLINT 到 BIGINT 进行隐式转换，但是不能从 BIGINT 到 SMALLINT 进行隐式转换。如果要完成这种转换，必须使用 CAST()。

（5）Impala 支持将字符串隐式转换成 TIMESTAMP。

（6）Impala 对 TIMESTAMP 数据类型和 from_unixtime()字符串使用固定的格式字符串抽取相应的内容。

（7）Impala 不存储本地时区时间，只存储相对于 GMT 的时间。

（8）Impala 针对超出列数据类型定义范围的溢出值返回的是一个该数据类型允许的边界值，而不是返回 NULL 值，因此我们自己捕获溢出值进行处理。比如，对于一个 TINYINT 的数据类型，它的范围是-128 到 127。如果我们为数据类型为 TINYINT 的列输入-200，将会返回-128，如果我们为其输入 200，则会返回 127。

（9）Impala 不提供虚拟列。

（10）Impala 不提供显示锁机制。

（11）Impala 不提供对某些属性显示配置。

4.11 将 SQL 移植到 Impala 上

即使Impala支持标准SQL，但是基于数据类型，内嵌函数，各厂商对SQL的不同扩展和Hadoop特定的语法差异，我们在将其他应用移植到 Impala 上时，仍然需要修改 SQL 源代码。即使 SQL 可以正常工作，基于性能的考虑，我们仍然需要对 SQL 进行进一步的修改。

1. 移植 DDL 和 DML 语句

当我们将 SQL 代码从传统数据库移植到 Impala 上时，在创建 schema 的过程中，我们会发现 DDL 有很多不同之处。比如，原有的 DDL 中涉及的物理文件，表空间，索引等在 Impala 中都无法对应。我们必须依据现有的分区规则，Hadoop 的文件格式等来重构 DDL 语句。

一般情况下，SQL 会与原有的系统保持较好的兼容性。但是，如果在原有系统中使用了厂商特有的语法或者特性，我们需要将其改写成标准 SQL。

因此，我们建议将原有系统中的 DDL 语句单独分离出来作为一个独立的安装脚本。这样，对于那些查询语句，我们可以放更多的精力来进行重用或者优化。

2. 数据类型迁移

（1）将 VARCHAR、VARCHAR2、CHAR 列修改为 STRING，并删除列声明中的长度限制。比如，我们需要将 VARCHAR(32)或者 CHAR(2) 修改为 STRING 类型。Impala 对字符串值的长度处理非常灵活，它可以将字符串限制为任何长度，但是并不会对字符数据本身进行额外的处理。

（2）对于像 NCHAR、NVARCHAR、NCLOB 等国际语言字符集类型的数据，虽然 Impala 支持作为 UTF8 字符集进行存储和查询，但是目前仍然有一些操作只有将其存储为 ASCII 字符集才能正常工作。

（3）将 DATE、DATETIME、TIME 等类型修改为 TIMESTAMP。对于时间的处理，我们不需要考虑时间精度问题，因为 TIMESTAMP 类型的精度都比其他的精度要高。确保应用程序逻辑和 ETL 过程处理的 TIMESTAMP 值是 UTC 时间，不能使用本地时区的时间。

在转换的过程中，我们可能会遇到要将日期或者时间的常量值或者字符串转换成 Impala 支持的类型。我们可以考虑使用 regexp_repalce()来对转换格式中的 YY，MM 或者 Impala 需要的前导符进行处理，将其转换成 Impala 兼容的格式。

（4）将 SYSDATE 替换为 NOW()。将原有日期值直接加减一个整数 N 来表示过去或者未来 N 天得某个时间点的情况，可以使用 INTERVAL 表达式，比如 NOW() + INTERVAL 30 DAYS。

（5）虽然 Impala 支持 INTERVAL 表达式，但是我们无法将其定义为表列的数据类型。对于任何的 INTERVAL 值，我们只能将其存为数字类型，再使用日期函数对其进行运算。比如，我们有一个表名称为 DEADLINE，其中有一个整形列 TIME_PERIOD，我们可以通过以下方式构建一个日期：

```
SELECT NOW() + INTERVAL time_period DAYS from deadlines;
```

（6）对于 YEAR 类型的列，我们可以使用 Impala 中的 SMALLINT 来代替。

（7）对于 DECIMAL 和 NUMBER 数据类型，如果没有精度要求，我们可以使用 FLOAT 或者 DOUBLE 代替。如果有精度要求，比如像金融数据，我们需要对表结构和应用逻辑做额外的处理。比如，对元的存储，我们可以将元和分分离开，分别使用整型数据存储，或者将数字类型以字符串方式存储，也可以使用 UDF 对数据进行处理。

（8）对于 FLOAD、DOUBLE、REAL 类型，在 Impala 中可以直接使用。使用时要删除原有的数据定义上的精度说明。事实上，在 Impala 内部，REAL 类型是 DOUBLE 的别名。将列声明为 REAL 类型和声明为 DOUBLE 类型是等效的。

（9）大多数其他系统的整型类型与 Impala 的是一致的，但是也可能存在定义相同名称不同的情况，比如 BIGINT 和 INT8。对于其他无法准确匹配数据表示范围的情况，我们需要将其转换为 Impala 中能够表示其范围的最小的整数类型。

（10）去掉其他系统的 SQL 中的 UNSIGNED 关键字，因为 Imapla 中的所有的数字类型都是无符号的。

（11）对于任何位运算的值，使用 Impala 中能够涵盖其范围的最小整型数据表示。

比如，对于 TINYINT 值，最大的正数为 127，而 8 位的二进制数的最大值为 255，所以用 TINYINT 无法表示，只能用 SMALLINT 来表示。

```
[hadoop-cs1:21000] > select cast(127*2 as tinyint);
+--------------------------+
| cast(127 * 2 as tinyint) |
+--------------------------+
| -2 |
+--------------------------+
[hadoop-cs1:21000] > select cast(128 as tinyint);
+----------------------+
| cast(128 as tinyint) |
+----------------------+
| -128 |
+----------------------+
[hadoop-cs1:21000] > select cast(127*2 as smallint);
+---------------------------+
| cast(127 * 2 as smallint) |
+---------------------------+
| 254 |
+---------------------------+
```

Impala 不支持使用形如 b’0101’来表示位元值。

（12）使用 STRING 类替代 CLOB 或者 TEXT 类型。而对于 BLOB，RAW

BINARY,VARBINARY 类型，目前 Impala 中并不支持。

（13）对于布尔类型可以使用 Impala 中的 BOOLEAN 代替。

（14）目前 Impala 不支持其他数据库支持的组合类型，嵌套类型及空间数据类型。我们可以将空间类型存成字符串，然后使用 UDF 来处理它。实践中，我们一般将不同的空间类型分别存放在不同的表中，以使这些数据可以很好地与非空间类型数据一起运算处理。

（15）去掉 SQL 中的 DEFAULT 子句。Impala 可以使用 Pig，Hive，MapReduce 作业等不同方式产生的数据文件。LOAD DATA 和外部表的快速导入机制意味着 Impala 灵活的支持不同格式的数据文件，而且 Impala 在进行查询之前不会对其中的数据进行检验和清洗。当我们通过 INSERT 语句将数据拷贝到 Impala 中时，我们可以使用像 CASE 或者 NVL 等条件函数来预处理 NULL 值。

（16）去掉 CREATE TABLE 或者 ALTER TABLE 等 SQL 中形如 PRIMARY KEY、FOREIGN KEY、UNIQUE、NOT NULL、UNSIGNED，或 CHECK 的约束。Impala 认为数据对以上约束的处理应该在 ETL 或者 ELT 过程中完成。当我们通过 INSERT 语句将数据拷贝到 Impala 中时，我们可以使用像 CASE 或者 NVL 等条件函数来预处理 NULL 值。

实践中，数据的校验和预处理应该在数据被加载进 Impala 之前完成。当然，如果数据已经被加载到了 Impala 中，我们可以通过 SQL 语句进一步校验列值是否在业务允许的范围内，是否为 NULL 等。但是这样做就相当于在 Impala 内部执行了一个类似于 ETL 的过程，而且处理过的数据要通过 INSERT…SELECT 将处理后的数据复制到一张新的表中。

（17）去掉形如 CREATE INDEX、DROP INDEX、ALTER INDEX 或者通过 ALTER TABLE 完成对索引操作的等效语句。去掉 CREATE TABLE、ALTER TABLE 中的 INDEX、KEY、PRIMARY KEY 等子句。因为 Impala 不支持索引，只是对数据仓库风格的批量读取进行了优化。

（18）对于传入的超出范围或者不正确的输入参数的内嵌函数，Impala 会返回 NULL 值，而不是抛出异常。当然，这是在查询选项 ABORT_ON_ERROR=true 时的效果。我们可以使用小批量的具有代表性的样本数据作为内嵌函数的输入来检查它是否可以返回预期的值。比如，对于不支持的 CAST 操作，将不会跑出异常：

```
select cast('foo' as int);
+--------------------+
| cast('foo' as int) |
+--------------------+
| NULL |
+--------------------+
```

（19）对于其他 Impala 不支持的数据类型，我们可以将其存储为字符串，并通过 UDF 来进行处理。

（20）我们可以通过在查询选项 ABORT_ON_ERROR=true 时，对数据进行初始化测试来确认数据文件中的数据类型 Impala 是否支持，或者是否可以转换为 Impala 支持的类型。如果在查询中使用了不允许的类型转换，这个查询选项 ABORT_ON_ERROR=ture 将会引起查询失败。

```
set abort_on_error=true;
select count(*) from (select * from t1);
```

如果数据文件中包含了不支持的转换格式，上述语句将会报错。

3. SQL 语句迁移

对于某些我们很熟悉的 SQL 语句，Impala 并不支持。

（1）Impala 中没有 DELETE 语句。Impala 使用在数据仓库的场景中，对数据的处理仅包括海量数据的转换或者移动。虽然不能使用 DELETE，但是我们可以使用 INSERT OVERWRITE 来替换整个表或者整个分区的数据，或者使用 INSERT…SELECT 来将数据拷贝到另一个表中。

（2）Impala 中没有 UPDATE 语句。与 DELETE 语句类似，Impala 适用的数据仓库场景也不会涉及到对数据的更新操作。为了保证数据不会进行更新操作，我们需要在数据进入 Impala 之前的 ETL 中进行处理，或者使用 INSERT…SELECT 来将数据从一个表拷贝到另一个指定文件格式或者分区类型的表中。

（3）Impala 中没有事务的概念，不支持 COMMIT、ROLLBACK 等。Impala 使用的类似于传统数据库中的 AUTOCOMMIT 的方式，对 Impala 中数据的改变会立即生效。

（4）Impala 中的数据库名称，表名，列名及其他命名不能与 Impala 保留的关键字冲突。相反，如果我们使用一个 Impala 不认识的关键字，那么这个关键字会被解释为表或者列别名。例如：SQL 语句 SELECT * FROM t1 NATURAL JOIN t2，Impala 不认识 NATURAL 关键字，根据语法分析，Impala 会将 NATURAL 解释为表 t1 的别名。如果我们遇到了不能返回预期结果的查询，我们需要检查在 JOIN 和 WHERE 中的名称是否在预留关键字列表中。

（5）Impala 支持 FROM 子句的子查询，但是不支持带 WHERE 子句的子查询。因此，我们不能使用像 WHERE column IN (subquery)这样的查询语句。另外，虽然 EXISTS 是 Impala 的预留关键字，但是 Impala 仍然不允许使用 EXISTS 或 NOT EXISTS 子句。

（6）Impala 支持 UNION 和 UNION ALL，但是不支持 INTERSECT。如果从业务上我们可以确认两个结果集根本不相交，或者合并后的结果集中出现重复值对业务和查询没有任何影响，那么 UNION ALL 比 UNION 更优越。因为 UNION ALL 避免了为了消除重复值而对整个结果集进行物化和排序。

（7）对于任何子查询，必须使用别名。

比如：

```
select count(*) from (select * from t1) contents_of_t1;
```

如果子查询 select * from t1 没有 contents_of_t1 的别名，将会报错。

（8）如果我们对查询中的一个表达式使用了别名，那么这个别名在同一个查询列表中不能重复出现。

```
select avg(x) as average, average+1 from t1 group by x;
```

```
ERROR: AnalysisException: couldn't resolve column reference: 'average'
```

在 Impala 中解决这个问题有两个方法，一个是将表达式重复编码，一个是使用 WITH 子句，在 WITH 子句中创建基于原始表查询的包含表达式别名的列。

解决方案 1：

```
select avg(x) as average, avg(x)+1 from t1 group by x;
```

解决方案 2：

```
with avg_t as (select avg(x) average from t1 group by x)
select average, average+1
from avg_t;
```

（9）Impala 不支持数据仓库场景中针对海量数据进行关联不太合适的连接类型。在某些情况下，Impala 支持某些连接类型，但是为了避免连接效率低下的问题需要明确的语法来指定。Impala 不支持自然连接或者反连接，但是可以通过 CROSS JOIN 操作符来进行笛卡尔乘积。

（10）Impala 不支持全部的分区类型。分区定义基于一个或者多个分区键的不重复的值的组合。即使创建分布式分区，Impala 也不会重新分布和校验数据。我们必须根据数据量和数据分布情况选择合适的分区键。对于使用了范围分区，列表分区，哈希分区或者按键值分区的表，在 Impala 中，我们使用 CREATE TABLE 和 ALTER TABLE 语句的分区语法。Impala 分区类似于每个范围都有确定值界定的范围分区，或者使用哈希函数为每一组键值生成一个总的键值分区。

（11）对于 TOP-N 查询，Impala 不是使用 ROWNUM 或者 ROW_NUM 的伪列，而是使用 LIMIT 子句。

4. 仔细检查 SQL 结构

应用中某些 SQL 在编码时考虑的是书写方便，而不是性能。有时候，自动生成的 SQL 语句通过 JDBC 或者 ODBC 直接执行可能效率极其低下，也可能执行时会超出 Impala 的执行限制。在我们迁移 SQL 代码时，要时刻保持警惕：

（1）不使用 STORED AS 子句的 CREATE TABLE 语句默认会创建纯文本文件格式，这种明文的文件格式对于异构系统的数据交换非常方便，但是对于海量数据的高性能查询却不是一个好的选择。

（2）不使用 PARTITION BY 子句的 CREATE TABLE 语句会在同一个位置存储所有的数据文件，这可能会带来由于数据量的不断骤增带来的扩展性问题。

另一方面，如果传统的数据库分区表在 Impala 中产生了大量的分区，而且每个分区数据量很小，将可能导致 Impala 无法充分利用并行查询的特性，从而使查询性能低下。

（3）INSERT…VALUES 语法可以对数据量非常小的表进行功能测试使用，通过这种方式每插入一行数据就会在 HDFS 上产生一个单独的小数据文件，非常不利于数据的海量扩展。如果一定要对数据进行变更，我们需要在数据进入 Impala 之前的过程进行处理，然后将 LOAD DATA 进

Impala 或者直接产生裸数据文件。这样数据无须转换处理就可以通过 Impala 直接访问，或者通过外部表的方式进行访问。

（4）如果我们的 ETL 过程没有针对 Hadoop 进行优化，那么我们可能最终产生很多碎片化的小数据文件，或者一个单个的大数据文件。无论哪种情况，都会导致 Impala 无法很好的利用并行查询和分区功能。在这种情况下，我们需要使用 INSERT…SELECT 语句将数据拷贝到一个新表，重新以更有效的方式组织数据。我们还可以使用 INSERT…SELECT 将数据写入效率更高的格式的文件中，或者将数据从非分区表写入分区表。

（5）Impala 中允许的表达式的数目比其他的数据库要少，这很可能导致非常复杂的查询（尤其是自动生成的 SQL 语句）执行失败。实践中，我们在 WHER 子句中使用的表达式的数目应该小于 2000 左右。

（6）业务允许的情况下，将 UNION 改写为 UNION ALL。

5. 校验语法语义，执行迁移

本节中，我们对迁移过程进行的操作将可能对语句的性能带来重大的影响。在确认了 SQL 语句迁移功能正确的前提下，我们要仔细检查 Schema 设计、物理文件分布等性能相关的方方面面，确保能够正确使用 Impala 并行机制，性能相关的 SQL 特性，以及与 Hadoop 其他组件很好的集成。

- 是否已对参与关联的表执行了 COMPUTE STATS 搜集统计信息？是否已对 INSERT ... SELECT 或者 CREATE TABLE AS SELECT 的源表执行了 COMPUTE STATS 搜集统计信息？
- 是否使用了符合数据规模，表结构和查询特点的文件格式？
- 是否有效的使用了分区？换句话说，是否能够通过分区列对 WHERE 条件进行很好的过滤？每个分区是否有足够的数据量以保证 Impala 的并行机制可以更好的发挥作用。
- ETL 过程是否产生了一定数量的有一定大小的文件，还是海量的小文件？

第 5 章

◂ Impala shell ▸

我们可以使用 Impala shell 工具（impala-shell）来创建数据库、表，插入数据，执行查询等。我们也可以通过在一个交互式会话中提交 SQL 语句来完成即席查询和钻取操作。为了使执行操作自动化，我们可以指定相应的命令行选项来处理单个的 SQL 语句或者脚本文件。imapla-shell 解释器支持所有在上一章中介绍的 SQL 语句，另外通过 shell 特有的命令行选项，我们也可以完成问题诊断和语句的优化。

在非交互模式中，查询会输出到标准输出 stdout 或者-o 选项指定的文件。如果查询错误，将会输出到标准错误 stderr。

在交互模式中，impala-shell 使用 readline 来编辑或重新调用之前的命令。

在 Cloudera Manager 4.1 或者更高版本中，impala-shell 将自动被安装到系统中。如果是在没有被 Cloudera Manager 管理的系统中，为了更好的和 Impala 进行交互，我们需要手动安装 impala-shell。

5.1 命令行选项

我们可以通过制定以下命令行选项来启动 impala-shell 来改变 shell 命令的执行方式。

（1）-B 或者--delimited：对查询结果去格式化，并将其按照指定的分隔方式进行分隔。在其他的 Hadoop 组件需要使用 Impala 产生的查询结果时，这个选项非常有用。这个选项避免了 Impala 对查询结果进行格式化的性能负载，尤其是在对海量数据进行压力测试时对性能提升效果更为明显。分隔符使用--output_delimiter 选项指定。-B 可以将查询结果输出到一个文件中。

示例：

```
-bash-4.1$ impala-shell -B
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.
```

```
(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > select * from student;
Query: select * from student
1       99      100     M
3       99      79      M
2       53      45      F
5       84      89      F
4       67      70      M
Returned 5 row(s) in 0.35s
```

（2）-o 或者--output_file：该选项后面跟文件名称，可以将输出结果存储为指定的文件。该选项经常与-q 选项一起使用，-q 指定查询的 SQL 语句，而本选项指定输出结果。在交互式查询中，我们可以看到在输出结果中会自动为每行添加一个行号，而事实上这个行号并不属于结果集本身。为了防止在使用-q -o 选项时这些额外信息输出，需要将标准错误 stderr 重定向到/dev/null。

示例：

```
-bash-4.1$ impala-shell -o student.txt
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > select * from student;
Query: select * from student
Returned 5 row(s) in 0.35s
[hadoop-cs1:21000] > exit
                   > ;
Goodbye
-bash-4.1$ cat student.txt
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 3  | 99      | 79   | M  |
```

```
| 5  | 84       | 89   | F  |
| 4  | 67       | 70   | M  |
| 2  | 53       | 45   | F  |
| 1  | 99       | 100  | M  |
+----+---------+------+----+
```

（3）--output_delimiter=：该选项对使用-B 选项去格式化输出的查询结果指定各字段间的分隔符。默认的分隔符为制表键（'\t'）。如果输出字段中包含了分隔符字符，这个字段将使用/进行转义。

示例：

```
-bash-4.1$ impala-shell -B --output_delimiter=',' -o student.txt
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT
2014)
[hadoop-cs1:21000] > select * from student;
Query: select * from student
Returned 5 row(s) in 0.35s
[hadoop-cs1:21000] > exit
                  > ;
Goodbye
-bash-4.1$ cat student.txt
3,99,79,M
2,53,45,F
1,99,100,M
5,84,89,F
4,67,70,M
-bash-4.1$
```

（4）-p 或者--show_profiles: 显示查询的执行计划（与 EXPLAIN 语句输出相同）和每个查询语句底层的执行步骤的详细信息。

```
-bash-4.1$ impala-shell -p
Starting Impala Shell without Kerberos authentication
```

```
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT
2014)
[hadoop-cs1:21000] > select * from student;
Query: select * from student
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 5  | 84      | 89   | F  |
| 4  | 67      | 70   | M  |
| 1  | 99      | 100  | M  |
| 3  | 99      | 79   | M  |
| 2  | 53      | 45   | F  |
+----+---------+------+----+
Query Runtime Profile:
Query (id=648acd0ddaca0bc:7a67ab1e0d20a89):
  Summary:
    Session ID: 7c48db8941756888:b1fbd5b5e901c6bd
    Session Type: BEESWAX
    Start Time: 2014-11-26 17:04:07.618098000
    End Time:
    Query Type: QUERY
    Query State: CREATED
    Query Status: OK
    Impala Version: impalad version 1.3.1-cdh5 RELEASE (build )
    User: impala
    Network Address: ::ffff:192.168.0.153:39169
    Default Db: default
    Sql Statement: select * from student
    Plan:
----------------
Estimated Per-Host Requirements: Memory=32.00MB VCores=1
WARNING: The following tables are missing relevant table and/or column statistics.
default.student
```

```
F01:PLAN FRAGMENT [PARTITION=UNPARTITIONED]
  01:EXCHANGE [PARTITION=UNPARTITIONED]
     hosts=3 per-host-mem=unavailable
     tuple-ids=0 row-size=27B cardinality=unavailable

F00:PLAN FRAGMENT [PARTITION=RANDOM]
  DATASTREAM SINK [FRAGMENT=F01, EXCHANGE=01, PARTITION=UNPARTITIONED]
  00:SCAN HDFS [default.student, PARTITION=RANDOM]
     partitions=1/1 size=51B
     table stats: unavailable
     column stats: unavailable
     hosts=3 per-host-mem=32.00MB
     tuple-ids=0 row-size=27B cardinality=unavailable
----------------
    ……
```

（5）-h 或者--help：显示帮助信息。

（6）-i 或者--imapad=：后面接主机名用来指定连接到指定的 impalad 节点。默认的连接端口为 21000。我们可以使用该命令连接到集群中任何一台 impalad 节点上。如果 Impala 启动时使用了备用端口，则需要使用--fe_port 选项标识。

示例：

```
-bash-4.1$ hostname
hadoop-cs1
-bash-4.1$ impala-shell -i hadoop-cs2
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs2:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs2:21000] >
```

（7）-q 或者--query=：该选项用于执行一个查询语句或者 shell 命令。Impala 不必等待查询或者 shell 执行结束即可返回。这里的查询仅限于像 SELECT、CREATE TABLE、SHOW TABLES 等单条的语句。因为 USE 语句是一条单独的 SQL 语句，而这个选项只能指定单条语句，所以如

果对象没有前缀只能访问default数据库，如果要访问其他数据库就必须指定数据库名称作为前缀。

```
-bash-4.1$ impala-shell -q 'select * from student'
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Query: select * from student
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 3  | 99      | 79   | M  |
| 2  | 53      | 45   | F  |
| 5  | 84      | 89   | F  |
| 4  | 67      | 70   | M  |
| 1  | 99      | 100  | M  |
+----+---------+------+----+
Returned 5 row(s) in 0.35s
-bash-4.1$
```

（8）-f 或者--query_file=：这个选项后面可以跟一个 SQL 查询脚本文件，脚本文件中的 SQL 必须使用“；”来分隔。

示例：

```
-bash-4.1$ cat query.sql
use default;
select * from student;
select mf,count(*) from student group by mf;
-bash-4.1$ impala-shell -f query.sql
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Query: use default
Query: select * from student
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 5  | 84      | 89   | F  |
| 4  | 67      | 70   | M  |
| 3  | 99      | 79   | M  |
| 2  | 53      | 45   | F  |
```

```
| 1  | 99       | 100  | M  |
+----+---------+------+----+
Returned 5 row(s) in 0.33s
Query: select mf,count(*) from student group by mf
+----+----------+
| mf | count(*) |
+----+----------+
| M  | 3        |
| F  | 2        |
+----+----------+
Returned 2 row(s) in 0.49s
```

（9）-k 或者--kerberos：该选项用来指定当 shell 连接到 impalad 节点时使用 kerberos 身份验证。但是如果 impalad 节点本身没有启用 kerberos，连接将会报错。

（10）-s 或者--kerberos_service_name：该选项后面跟 kerberos 服务名称让 impala-shell 验证一个特定的 impalad 服务。如果没有指定 kerberos 服务名称，将使用 impala 作为默认的名称。如果该选项用于一个不支持 kerberos 的连接，将会返回错误。

（11）-V 或者--verbose：启用详细信息输出。

（12）--quiet：禁用详细信息输出。

```
-bash-4.1$ impala-shell --quiet
Starting Impala Shell without Kerberos authentication
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
[hadoop-cs1:21000] > select * from student;
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 3  | 99       | 79   | M  |
| 2  | 53       | 45   | F  |
| 5  | 84       | 89   | F  |
| 1  | 99       | 100  | M  |
| 4  | 67       | 70   | M  |
+----+---------+------+----+
[hadoop-cs1:21000] >
```

（13）-v 或者--version：显示版本信息。

（14）-c ：如果查询失败继续执行。

（15）-r 或者--refresh_after_connect：连接后刷新 Impala 元数据信息。效果和连接之后执行 REFRESH 语句相同。

（16）-d 或者--database=：该选项后面跟上数据库的名称用于连接到指定的数据库。该选项的效果和连接上之后执行 USE 的效果相同。如果不指定该选项，将连接到系统默认的 default 数据库。

```
-bash-4.1$ impala-shell -d d1
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 2014)
Query: use d1
[hadoop-cs1:21000] > select current_database();
Query: select current_database()
+--------------------+
| current_database() |
+--------------------+
| d1                 |
+--------------------+
Returned 1 row(s) in 0.10s
```

（17）-l：该选项用于启用 LDAP 身份验证。

（18）-u：如果启用了-l 选项，则用-u 来指定身份验证的用户名称。连接时 shell 将会提示输入密码。

本节中的大部分命令行选项可以用于将 Impala 的查询写入 shell 脚本中。比如，通过-f 选项我们可以将要执行的语句写入文件，通过该选项调用要查询的文件；通过-B -o --output_delimiter 可以将查询结果以特定的分隔符写入文件供其他 Hadoop 组件调用。

5.2 连接到 Impalad

在 Impala 中，我们只有连接到 impalad 进程节点后才能执行查询操作。当我们使用 impala-shell 连接到 impalad 进程节点时可以配置相关的命令行选项指定连接信息。我们可以连接到任何一个运行了 impalad 进程的 DataNode，接受了连接的节点将作为协调者节点运行查询任务。

简单起见，我们可以总是连接到同一个 impalad 节点。如果在本地节点执行 impala-shell 进行连接，我们可以把主机名指定为 localhost。当我们通过一个节点执行操作导致表数据或者元数据更新时，通过另一个节点连接后必须执行 REFRESH 操作刷新元数据信息。如果我们始终连接到同一个节点，在这个节点上进行操作将能有效地避免频繁的执行 REFRESH 操作。

从负载均衡和灵活性的角度考虑，我们可以连接到任何一个 impalad 节点上执行查询。在这种情况下，如果表数据或者元数据信息被其他节点更新，该连接将会根据情况对我们查询的表执行 REFRESH table_name 或者执行 REFRESH 刷新全部表的元数据信息。

为了让 imapla-shell 连接到任何 impalad 节点：

1. 启动 impala-shell

```
$ impala-shell
```

执行该命令后，将会看到如下提示信息：

```
Starting Impala Shell without Kerberos authentication
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.3.1-cdh5 () built on Mon Jun  9 09:30:26 PDT 
2014)
[Not connected] >
```

2. 使用 connect 连接到 Impala 实例

```
[Not connected] > connect hadoop-cs1;
Connected to hadoop-cs1:21000
Server version: impalad version 1.3.1-cdh5 RELEASE (build )
[hadoop-cs1:21000] >
```

5.3 运行命令

我们可以通过按两次 TAB 键获得所有命令的列表：

```
[hadoop-cs1:21000] >
alter      create     describe   exit       help       insert     profile    select
shell      unset      values     with
connect    desc       drop       explain    history    load       quit       set        show
use        version
[hadoop-cs1:21000] >
```

可以在这个 shell 中直接执行 SQL 语句，演示如下：

```
[hadoop-cs1:21000] > select * from student limit 3;
Query: select * from student limit 3
+----+---------+------+----+
| id | english | math | mf |
+----+---------+------+----+
| 5  | 84      | 89   | F  |
| 1  | 99      | 100  | M  |
| 4  | 67      | 70   | M  |
+----+---------+------+----+
Returned 3 row(s) in 0.35s
[hadoop-cs1:21000] >
```

5.4 命令参考

当 shell 连接到 Impalad 节点之后，我们可以使用下述命令向 Impala 发送请求。有两种发送请求的方式，一种是直接在 shell 命令提示符中交互式执行，一种是通过启动 shell 时指定-q 选项。这些命令中大多数都是 SQL 语句，具体 SQL 的语法请参考上一章。

- alter: 改变一个 Impala 表，Hive 表的表结构或者配置信息。具体的命令有 ALTER TABLE 和 ALTER VIEW 语句。
- compute stats: 搜集表性能相关的统计信息，在优化 Impala 查询时使用。
- connect: 连接到 Impalad 节点。连接的默认端口号为 21000，如果要使用备用端口，则需要指定--fe_port 标识。SET 命令只有连接到 Impalad 节点之后才会生效。
- describe: 显示指定表的列，列数据类型，列注释信息等。DESCRIBE FORMATTED 命令

还会显示 HDFS 数据目录，分区，表内部属性等更详细的信息。另外，我们也可以把这个命令缩写为 DESC。

- drop：删除一个对象，在某些情况下还会删除相关的数据文件。相关的命令有 DROP TABLE、DROP VIEW、DROP DATABASE、DROP FUNCTION 等。
- explain：显示一个查询的执行计划及它执行的各个步骤信息。这个的步骤信息可能是 MAP/REDUCE 的各阶段，元数据操作，文件系统操作等。
- help：显示所有可用的命令和选项的列表。
- history：维护一个会话无关的执行过的命令的历史记录。这个历史记录存储在 ~/.impalahistory 文件中。
- insert：向指定的表中写入数据。这里的写入可能是向已存在数据的表中插入数据，也可能是覆盖现有表中的数据。
- invalidate metadata：更新元数据信息。在创建、删除、修改了数据库、表、分区后使用本命令。
- profile：显示最近的查询到底层信息用于问题诊断或者性能优化。
- quit：退出 shell。quit 命令也需要以“；”结尾，shell 才能识别。
- refresh：刷新 HDFS 数据文件位置相关的元数据信息。在向表中加载了新的数据文件之后，需要使用该命令。
- select：查询满足特定条件的数据集。select 返回的所有的数据集既可以在控制台输出，也可以输出到文件。
- set：管理 impala-shell 会话的查询选项。这些选项用来做问题诊断或者优化查询。执行没有任何参数的 SET 选项，将返回所有选项的当前值。这个当前值可能是 impalad 进程的默认值，也可能是启动 impalad 时指定的值，或者是在这个会话中刚刚设置的某个参数值。如果要修改某个选项，使用 SET OPTION=VALUE 的语法，这里的 value 可以使用布尔值、数字或者字符串。如果要恢复默认值，可以使用 unset 命令。
- shell：不退出 impala-shell 执行操作系统的命令。我们也可以使用！作为 shell 命令的缩写。
- show：显示对象的元数据信息。该命令还可以用来搜集特定的数据库或者表的相关信息。
- unset：用于删除我们通过 set 命令设置的查询选项，将其修改为默认值。
- use：切换到指定的数据库。切换到指定的数据库之后，我们在访问该数据库中的对象时就不必再指定数据库名称作为前缀。
- version：返回 Impala 的版本信息。

5.5 查询参数设置

我们可以在 impala-shell 中指定如下参数，修改后的参数将对这个会话中执行的所有查询有效。某些选项用于在日常操作中提高 Impala 的可用性、性能和灵活性。另外一些选项用于在故障

诊断或者调试时对 Impala 进行特殊的控制。

（1）ABORT_ON_DEFAULT_LIMIT_EXCEEDED

该参数与 DEFAULT_ORDER_BY_LIMIT 结合使用，确保 ORDER BY 查询的结果不会被截断。如果一个没有使用 LIMIT 的 ORDER BY 语句结果集超过了 DEFAULT_ORDER_BY_LIMIT 的值，这个查询将会被中断，而不是返回一个没有处理完的不正确的结果。

类型：布尔型。

默认值：FALSE。

当指定了 LIMIT 子句时，DEFAULT_ORDER_BY_LIMIT 和 ABORT_ON_DEFAULT_LIMIT_EXCEEDED 参数不会有任何影响：

```
[hadoop-cs1:21000] > select x from three_rows order by x limit 5;
Query: select x from three_rows order by x limit 5
Query finished, fetching results ...
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.27s
如果只指定 DEFAULT_ORDER_BY_LIMIT 参数，结果集会被截断：
[hadoop-cs1:21000] > set default_order_by_limit=5;
DEFAULT_ORDER_BY_LIMIT set to 5
[hadoop-cs1:21000] > select x from ten_rows order by x;
Query: select x from ten_rows order by x
Query finished, fetching results ...
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
Returned 5 row(s) in 0.30s
```

如果指定了 ABORT_ON_DEFAULT_LIMIT_EXCEEDED 参数，查询将会被取消。

```
[hadoop-cs1:21000] > set abort_on_default_limit_exceeded=true;
ABORT_ON_DEFAULT_LIMIT_EXCEEDED set to true
[hadoop-cs1:21000] > select x from ten_rows order by x;
Query: select x from ten_rows order by x
Query aborted, unable to fetch data
Backend 0:DEFAULT_ORDER_BY_LIMIT has been exceeded.
```

（2）ABORT_ON_ERROR

如果未启用该选项，当某个 Impalad 进程节点故障时，已经发出的查询会在其他节点继续执行，并返回一个依赖部分 Impalad 进程节点运算的结果。这个结果仅依赖于正常运行的 Impalad 进程节点上的数据进行计算，所以计算出的结果可能不完全正确，至少是不完整的。当启用这个选项之后，如果任何节点发生错误，Impala 将立即中断查询。这个选项可以最大限度的帮助我们搜集错误发生时的诊断信息。通过诊断信息我们可以确认是否发生的是之前同样的问题，是在所有节点还是只在一个节点上发生的。目前，Impala 可以忽略的错误包括像列类型不匹配的数据损坏等。

当 ABORT_ON_ERROR 为 OFF 时，为了控制 Impala 产生的非关键的错误信息的多少，需要使用 MAX_ERRORS 选项。

类型：布尔。

默认值：FALSE。

（3）ALLOW_UNSUPPORTED_FORMATS

该选项是一个过时的将要被抛弃的选项，在早期用于支持不同的文件格式。在新版本中，该选项会被彻底删除。

类型：布尔。

默认值：FALSE。

（4）BATCH_SIZE

SQL 操作符一次可以计算的行数。如果未指定该值或者指定为 0，将使用默认值 1024。

默认值：0（也就是 1024）。

（5）DEBUG_ACTION

用于 Cloudera 内部故障诊断和调试。

类型：STRING。

默认值：空字符串。

（6）DEFAULT_ORDER_BY_LIMIT

为了避免对海量的结果集进行排序，Impala 在查询时如果使用了 ORDER BY 则必须使用 LIMIT 子句限定返回的数据条数。对一个海量数据结果集进行排序是一个特别消耗内存的操作，

只有在对所有的数据排序完成之后，才能返回结果。

为了避免对已有的应用修改源代码，对有 ORDER BY 的 SQL 加入 LIMIT 子句，我们可以通过设置 DEFAULT_ORDER_BY_LIMIT 参数设定允许排序之后返回的最大记录行数。比如，如果设定 DEFAULT_ORDER_BY_LIMIT=10，查询将会返回 TOP-10 结果。或者我们可以设置一个很大的值，比如 DEFAULT_ORDER_BY_LIMIT=1000000，用于对返回的记录行数进行检查确认。当然，如果返回的行数小于 1000000，这个选项对查询没有任何影响。

该参数默认值是-1，表示不设定返回结果的上限值。如果在所有的包含 ORDER BY 的 SQL 中都指定了 LIMIT 子句，可以使用该默认值。

示例：

如果 SQL 指定了 LIMIT 子句，则 DEFAULT_ORDER_BY_LIMIT 参数不起作用：

```
[hadoop-cs1:21000] > select x from three_rows order by x limit 5;
Query: select x from three_rows order by x limit 5
Query finished, fetching results ...
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.27s
```

如果指定了 DEFAULT_ORDER_BY_LIMIT 参数，则结果集会被截断：

```
[hadoop-cs1:21000] > set default_order_by_limit=5;
DEFAULT_ORDER_BY_LIMIT set to 5
[hadoop-cs1:21000] > select x from ten_rows order by x;
Query: select x from ten_rows order by x
Query finished, fetching results ...
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
```

```
Returned 5 row(s) in 0.30s
```

（7）DISABLE_CODEGEN

这是一个用于诊断系统崩溃的调试参数。如果查询由于“illegal instruction”或者其他特定的硬件错误导致查询失败，我们可以设置 DISABLE_CODEGEN=true，再次运行有问题的查询。如果查询只有在DISABLE_CODEGEN=true时才能运行成功，我们需要向厂商提交详细的问题报告。在其他任何情况下，不得将该参数置为 true。因为将其置为 true 后，会为每个执行的查询增加一定额外的开销，导致性能问题。如果对一个有很多对小表访问的短查询的系统上将该参数置为 true，将很快达到系统的最大吞吐量。

类型：布尔。

默认值：FASLE。

（8）EXPLAIN_LEVEL

该参数控制 EXPLAIN 语句输出的信息量。执行计划的基本输出信息可以帮助我们从顶层分析像数据量过大或者分区过多等性能问题。而更详细的信息则可以显示中间结果是如何在各节点之间交互，以及像 ORDER BY,GROUP BY,JOIN,WHERE 等 SQL 如何在分布式系统中的实现等。

类型：STRING 或 INT。

默认值：1。

参数：参数值的范围为数字 0~3。

- 0：即 MINIMAL，是执行计划的最小输出，包含执行计划最简要的信息，每个操作占用一行。使用该参数主要是用来检查一个长 SQL 中连接的顺序是否符合预期。
- 1：即 STANDARD，是执行计划的标准输出，也是默认的输出，包含分布式查询执行的逻辑路径。
- 2：即 EXTENDED，是执行计划的扩展输出，包括查询优化器如何使用统计信息。通过该信息我们可以了解一个查询如何通过搜集统计信息，添加 Hint，或者删除谓词进行优化。
- 3：即 VERBOSE，是执行计划最详细的输出，包括一个查询是如何被分配到各节点上，并通过管道进行连接等底层信息。查看这个级别的信息主要是为了从底层对 Impala 自身进行性能优化，而不是从用户级别对 SQL 进行改写。

改变这个参数值可以控制执行计划输出的信息量。当我们需要确认查询是否是以我们预期的分布式的方式执行，或者想了解像大表关联、分区过多、向 Parquet 表插入数据等对资源消耗特别大的操作如何执行时，可以选择使用参数 2 或者参数 3。当我们使用了准入控制或者资源管理时，使用执行计划的扩展的信息可以计算资源使用率。

注意事项：我们需要自底向上阅读执行计划。执行计划最下面的行显示了查询的初始化工作，包括扫描了哪些数据文件等；中间的部分展示了中间结果是如何从一个节点传送到另一个节点；最上面的部落则展示了从其他节点的结果返回给协调者节点并形成最终结果。

执行计划最左侧一列在执行计划生成时由 Impala 底层产生，它的顺序并不代表操作执行的顺序。如果这个顺序错乱，对执行计划本身并无影响。

如果统计信息缺失，在任何级别的执行计划里都有一个警告。此时需要使用 COMPUTE STATS 搜集统计信息，消除警告。

在 impala-shell 中使用 PROFILE 命令可以显示详细的执行计划，显示的级别与 EXPLAIN_LEVEL=3 相同。

示例：

本示例使用一个简单的空表来演示执行计划的重要的组成部分：

```
[hadoop-cs1:21000] > create table t1 (x int, s string);
[hadoop-cs1:21000] > set explain_level=1;
[hadoop-cs1:21000] > explain select count(*) from t1;
+-----------------------------------------------------------------------------
-------+
| Explain String
|
+-----------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=10.00MB VCores=1
|
| WARNING: The following tables are missing relevant table and/or column statistics.
|
| explain_plan.t1
|
|
|
| 03:AGGREGATE [MERGE FINALIZE]
|
| | output: sum(count(*))
|
| |
|
| 02:EXCHANGE [PARTITION=UNPARTITIONED]
|
| |
|
| 01:AGGREGATE
|
| | output: count(*)
|
```

```
| |
|
| 00:SCAN HDFS [explain_plan.t1]
|
| partitions=1/1 size=0B
|
+------------------------------------------------------------------------------
-------+
[hadoop-cs1:21000] > explain select * from t1;
+------------------------------------------------------------------------------
-------+
| Explain String
|
+------------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0
|
| WARNING: The following tables are missing relevant table and/or column statistics.
|
| explain_plan.t1
|
|
|
| 01:EXCHANGE [PARTITION=UNPARTITIONED]
|
| |
|
| 00:SCAN HDFS [explain_plan.t1]
|
| partitions=1/1 size=0B
|
+------------------------------------------------------------------------------
-------+
[hadoop-cs1:21000] > set explain_level=2;
[hadoop-cs1:21000] > explain select * from t1;
+------------------------------------------------------------------------------
-------+
| Explain String
|
```

```
+------------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0
|
| WARNING: The following tables are missing relevant table and/or column statistics.
|
| explain_plan.t1
|
|
|
| 01:EXCHANGE [PARTITION=UNPARTITIONED]
|
| | hosts=0 per-host-mem=unavailable
|
| | tuple-ids=0 row-size=19B cardinality=unavailable
|
| |
|
| 00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM]
|
| partitions=1/1 size=0B
|
| table stats: unavailable
|
| column stats: unavailable
|
| hosts=0 per-host-mem=0B
|
| tuple-ids=0 row-size=19B cardinality=unavailable
|
+------------------------------------------------------------------------------
-------+
[hadoop-cs1:21000] > set explain_level=3;
[hadoop-cs1:21000] > explain select * from t1;
+------------------------------------------------------------------------------
-------+
| Explain String
|
+------------------------------------------------------------------------------
```

```
-------+
    | Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0
    |
    | WARNING: The following tables are missing relevant table and/or column statistics.
    |
    | explain_plan.t1
    |
    |
    |
    | F01:PLAN FRAGMENT [PARTITION=UNPARTITIONED]
    |
    | 01:EXCHANGE [PARTITION=UNPARTITIONED]
    |
    | hosts=0 per-host-mem=unavailable
    |
    | tuple-ids=0 row-size=19B cardinality=unavailable
    |
    |
    |
    | F00:PLAN FRAGMENT [PARTITION=RANDOM]
    |
    | DATASTREAM SINK [FRAGMENT=F01, EXCHANGE=01, PARTITION=UNPARTITIONED]
    |
    | 00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM]
    |
    | partitions=1/1 size=0B
    |
    | table stats: unavailable
    |
    | column stats: unavailable
    |
    | hosts=0 per-host-mem=0B
    |
    | tuple-ids=0 row-size=19B cardinality=unavailable
    |
    +-----------------------------------------------------------------------------
-------+
```

在警告信息中显示了 Impala 为生成执行计划所需要的信息。此处，我们需要搜集表的统计信

息：

```
[hadoop-cs1:21000] > compute stats t1;
+-----------------------------------------+
| summary |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
[hadoop-cs1:21000] > explain select * from t1;
+----------------------------------------------------------------------------+
| Explain String |
+----------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0 |
| |
| F01:PLAN FRAGMENT [PARTITION=UNPARTITIONED] |
| 01:EXCHANGE [PARTITION=UNPARTITIONED] |
| hosts=0 per-host-mem=unavailable |
| tuple-ids=0 row-size=20B cardinality=0 |
| |
| F00:PLAN FRAGMENT [PARTITION=RANDOM] |
| DATASTREAM SINK [FRAGMENT=F01, EXCHANGE=01, PARTITION=UNPARTITIONED] |
| 00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM] |
| partitions=1/1 size=0B |
| table stats: 0 rows total |
| column stats: all |
| hosts=0 per-host-mem=0B |
| tuple-ids=0 row-size=20B cardinality=0 |
+----------------------------------------------------------------------------+
```

对于连接查询，或者其他的复杂查询，我们需要根据具体的情况确定执行计划的输出级别，并检查输出的执行计划信息。如下示例显示了一个三表连接的 SQL 输出的执行计划，然后使用[SHUFFLE]的 Hint 将前两个表的连接方式从 BROADCAST 改为 SHUFFLE。

```
[hadoop-cs1:21000] > set explain_level=1;
[hadoop-cs1:21000] > explain select one.*, two.*, three.* from t1 one, t1 two, t1
three
where one.x = two.x and two.x = three.x;
+--------------------------------------------------------------------------
-------+
| Explain String
```

```
|
+-----------------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3
|
|
|
| 07:EXCHANGE [PARTITION=UNPARTITIONED]
|
| |
|
| 04:HASH JOIN [INNER JOIN, BROADCAST]
|
| | hash predicates: two.x = three.x
|
| |
|
| |--06:EXCHANGE [BROADCAST]
|
| | |
|
| | 02:SCAN HDFS [explain_plan.t1 three]
|
| | partitions=1/1 size=0B
|
| |
|
| 03:HASH JOIN [INNER JOIN, BROADCAST]
|
| | hash predicates: one.x = two.x
|
| |
|
| |--05:EXCHANGE [BROADCAST]
|
| | |
|
| | 01:SCAN HDFS [explain_plan.t1 two]
|
```

```
| | partitions=1/1 size=0B
|
| |
|
| 00:SCAN HDFS [explain_plan.t1 one]
|
| partitions=1/1 size=0B
|
+-----------------------------------------------------------------------------
-------+
[hadoop-cs1:21000] > explain select one.*, two.*, three.* from t1 one join [shuffle]
t1
two join t1 three where one.x = two.x and two.x = three.x;
+-----------------------------------------------------------------------------
-------+
| Explain String
|
+-----------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3
|
|
|
| 08:EXCHANGE [PARTITION=UNPARTITIONED]
|
| |
|
| 04:HASH JOIN [INNER JOIN, BROADCAST]
|
| | hash predicates: two.x = three.x
|
| |
|
| |--07:EXCHANGE [BROADCAST]
|
| | |
|
| | 02:SCAN HDFS [explain_plan.t1 three]
|
```

```
| | partitions=1/1 size=0B
|
| |
|
| 03:HASH JOIN [INNER JOIN, PARTITIONED]
|
| | hash predicates: one.x = two.x
|
| |
|
| |--06:EXCHANGE [PARTITION=HASH(two.x)]
|
| | |
|
| | 01:SCAN HDFS [explain_plan.t1 two]
|
| | partitions=1/1 size=0B
|
| |
|
| 05:EXCHANGE [PARTITION=HASH(one.x)]
|
| |
|
| 00:SCAN HDFS [explain_plan.t1 one]
|
| partitions=1/1 size=0B
|
+-------------------------------------------------------------------------------
-------+
```

对于一个涉及多个表的连接查询，默认输出的执行计划可能包含很多信息，而我们真正关注的可能是表的连接顺序，或者 JOIN 的方式（BROADCAST 或者 SHUFFLE）。如果我们只有这个需求，可以把执行计划的显示级别修改为 0。如下示例显示了前两个表如何被 HASH 分布到各节点上，又将第三个表 BROADCAST 到所有节点上进行处理的过程。

```
[hadoop-cs1:21000] > set explain_level=0;
[hadoop-cs1:21000] > explain select one.*, two.*, three.* from t1 one join [shuffle]
t1
```

```
two join t1 three where one.x = two.x and two.x = three.x;
+----------------------------------------------------------+
| Explain String |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3 |
| |
| 08:EXCHANGE [PARTITION=UNPARTITIONED] |
| 04:HASH JOIN [INNER JOIN, BROADCAST] |
| |--07:EXCHANGE [BROADCAST] |
| | 02:SCAN HDFS [explain_plan.t1 three] |
| 03:HASH JOIN [INNER JOIN, PARTITIONED] |
| |--06:EXCHANGE [PARTITION=HASH(two.x)] |
| | 01:SCAN HDFS [explain_plan.t1 two] |
| 05:EXCHANGE [PARTITION=HASH(one.x)] |
| 00:SCAN HDFS [explain_plan.t1 one] |
+----------------------------------------------------------+
```

（9）HBASE_CACHE_BLOCKS

设置该参数相当于在 HBase Java 应用程序中调用了 org.apache.hadoop.hbase.client.Scan 的 setCacheBlocks 方法，可用于控制 HBase region 服务器的内存压力。该参数经常与 HBASE_CACHING 结合使用。

类型：BOOLEAN。

默认值：FASLE。

（10）HBASE_CACHING

设置该参数相当于在 HBase Java 应用程序中调用了 org.apache.hadoop.hbase.client.Scan 的 setCaching 方法，可用于控制 HBase region 服务器的内存压力。该参数经常与 HBASE_CACHE_BLOCKS 结合使用。

类型：BOOLEAN。

默认值：0。

（11）MAX_ERRORS

在 Impala 日志文件中可记录的某个查询的非关键性错误的最大个数。如果是一个有十亿行记录的表，每行产生一个非关键性错误，如果不限制将会产生十亿个非关键性错误，这些信息都会被一一记录下来。如果指定为 0 或者不指定，则使用默认值 1000。

这个参数可以控制多少个错误被记录下来。如果我们想在 Impala 遇到错误的时候中断查询，可以使用 ABORT_ON_ERROR 参数。

默认值：0（也就是 1000）。

（12）MAX_IO_BUFFERS

该参数将被抛弃。当前如何设置没有影响。

默认值：0。

（13）MAX_SCAN_RANGE_LENGTH

该参数表示扫描范围的最大长度。该参数和表的 HDFS 块数共同决定集群将使用多少个 CPU 核来处理查询请求。

该参数越小，就越增加并行度，要使用更多的 CPU 核进行运算。但是如果过小，也可能会带来性能问题，因为将任务拆分到多颗 CPU 上本身也需要负载。

该参数只对 HDFS 表起作用，对 Parquet 表无影响。如果指定为 0 或者为指定该参数，其大小与每张表的 HDFS 块大小相同。通常情况下，对于大多数文件格式来说，这个值为几 MB，而对于 Parquet 表，这个值为 1GB。

虽然这个扫描的范围我们可以任意定义，但是在底层 Impala 使用 8MB 的读缓冲区，这样可以保证查询时不必为表分配和表同等大小的内存。

默认值：0。

（14）MEM_LIMIT

如果没有启用资源管理功能，该参数用来定义每个节点的最大内存数量。如果在查询过程中，任何一个节点超过了这个限制，Impala 将会自动终止查询。在查询执行的过程中，Impala 会定期检查各节点的内存是否超出了限制。即使真正使用的内存超出了该参数的限制，只要在 Impala 检查时没有超过，这个查询就不会被终止。

在 CDH5 中，如果启用了资源管理功能，则这个参数的表现与上述机制不同。如果设置了该参数，它的值将覆盖 Impala 自动计算的内存值。Impala 为每个节点从 YARN 请求该参数指定的内存数，只有有足够的内存可用，查询才会继续进行，虽然事实上可能使用的内存比请求的少。此时，这个参数成为查询可以正常执行的硬性限制。如果启用了资源管理功能，但是没有设置 MEM_LIMIT 参数，Impala 将为每一个查询计算每个节点需要的内存数量，然后从 YARN 请求相应大小的内存，使用相应的大小初始化 MEM_LIMIT 参数。在这种情况下，如果查询实际需要的内存比 Impala 计算的内存大，Impala 就认为内存超出过 MEM_LIMIT 的限制，终止查询执行。

默认值：0。

（15）NUM_NODES

仅在调试过程中限制节点数时使用。该参数仅接受两个参数，一个是 0，表示所有节点均参与运算，一个是 1，表示所有的运算都在协调者节点上进行。如果我们怀疑一个查询的多个节点的交互有问题，可以将参数设置为 1，让其在一个节点上执行来排除该问题。

默认值：0。

（16）NUM_SCANNER_THREADS

一个查询在每个节点上扫描线程的最大数量。默认情况下，节点有多少个 CPU 核就会启动多少个扫描线程。对于一个有一定负载的集群，我们可以适当的改下这个参数。Impala 默认使用的

就是最大值，所以将值改大对 Impala 没有任何影响。

默认值：0。

（17）PARQUET_COMPRESSION_CODEC

当 Impala 向 Parquet 表插入数据时，底层的压缩算法由该参数控制。该参数允许的值包括：snappy，gzip，none。这个参数是大小写敏感的。

如果该参数设置为一个 Impala 无法识别的值，所有的查询都会报错。

默认值：SNAPPY。

（18）PARQUET_FILE_SIZE

在向 Impala 的 Parquet 表插入数据时，该参数指定 Parquet 数据文件的最大大小。对于小表或者分区表，默认的 Parquet 块大小为 1GB。如果我们想增大并行度，让数据尽可能的分布在多个节点上，可以减少这个参数的值。减小这个值的大小也可以降低数据写入磁盘时的内存压力。

默认值：0（默认为 1GB）。

（19）REQUEST_POOL

查询需要提交的池或者队列的名称。该参数只有在启用了 Impala 准入控制特性或者基于 YARN 的资源管理时有效。

在 CDH5 中，Impala 使用的默认名称为 YARN_POOL。

默认值：空。

（20）RESERVATION_REQUEST_TIMEOUT

Impala 为一个完全授权或者拒绝授权预留的最大毫秒数。

默认值：300000（5 分钟）。

（21）SUPPORT_START_OVER

保持该值为 FALSE。

默认值：FALSE。

（22）SYNC_DDL

启用了该参数将会导致任何 DDL 语句必须通过 Catalog 服务将更新的元数据信息推送到所有节点上之后，才会返回。所以，如果后续我们通过 CONNECT 连接到某个节点上时，因为它已经获得了最新的元数据信息，我们就不用担心识别不到新创建的对象的情况。

虽然 INSERT 语句是一个典型的 DML 语句，但是在 Impala 中，SYNC_DDL 仍然能够起作用。如果启用了 SYNC_DDL 参数，INSERT 语句只有在将所有的元数据变化推送到所有 Impala 节点上后才算完成。在 Impala 底层，INSERT 语句与传统的数据库中的 DDL 语句类似，因为它涉及到像为新的数据文件创建 HDFS 块位置信息，或者为分区表增加新的分区等元数据信息的变化。

默认值：FALSE。

（23）V_CPU_CORES

从 YARN 中请求的每台主机的虚拟 CPU 核的数量。如果设置了该参数，Impala 将使用该值覆盖 Impala 自动计算的值。使用该参数，可以将 CDH5 和 Impala 的资源管理特性连接在一起。

默认值：0（使用自动计算的值）。

第 6 章

◂ Impala管理 ▸

作为一个管理员，我们需要监控 Impala 的资源使用情况，进行必要的维护操作以保持 Impala 能够与其他 Hadoop 组件协同工作。为了解决已有的问题或者潜在的问题，我们可能需要重新配置 Impala 或者其他 Hadoop 组件。

作为一个管理员，我们的任务包括安装，升级，配置 Impala 集群。如果有安全方面的需求，我们还需要对 Impala 的安全策略进行配置。另外，我们还要协调 Impala 与其他 Hadoop 组件协同工作。

6.1 准入控制和查询队列

准入控制是 Impala 的一个功能，为避免对一个繁忙的 CDH 集群产生过多的内存压力，它对并发执行的 SQL 查询进行强制限制。如果我们的 Impala 集群有时空载，有时负载过重，我们可以启用这个功能。如果将在一个负载很轻的集群上可以执行的语句放到一个负载很重的集群上执行，这个语句可能因为内存不足而被终止执行。准入控制功能为高并发查询避免内存不足提供了有利的保障。

6.1.1 准入控制概述

在一个繁忙的 CDH 集群上，我们可以通过不断的实验找到一个并发执行的查询的最佳数目。因为 Impala 查询都是对 IO 消耗很严重的，如果整个集群的 IO 能力已经达到极限，我们再提交新的查询，这时不但新的查询执行会产生瓶颈，由于它对集群 IO 资源的消耗会导致其他的查询也会受到影响。Impala 默认会终止那些超过了指定内存限制的查询任务，所以如果我们一次提交了几个很大的查询，可能我们不得不对由于内存限制而被 Impala 取消的查询重新执行。

准入控制功能可以让我们在集群侧对并发执行的查询的数目和使用的内存设置一个上限。那些超过限制的查询不会被取消，而是被放在队列中等待执行。一旦其他的查询执行结束释放了相关资源，队列中的查询任务就可以继续执行。

6.1.2 准入控制和 YARN

准入控制功能在一定程度上和 YARN 资源管理框架很类似，他们可以分别单独使用，也可以一起使用。本部分内容将介绍它们的差异，为我们以何种方式来管理资源提供帮助。

准入控制是一个轻量级的去中心化的系统，它适用于系统的主要负载来自于 Impala 查询的集群。它以软限制的方式控制 Impala 的内存被相对稳定的方式使用，而不是以要么执行要么取消的方式限制资源的使用。它可以运行在 CDH4 或者 CDH5 上。

因为准入控制不了解像 MapReduce 作业这样的、Impala 之外的其他 Hadoop 组件的负载情况，我们需要使用 YARN 静态服务池来管理被 Impala 和其他 Hadoop 组件共享的资源。我们可以将一定百分比的资源分配给 Impala 使用，而将剩余的资源分配给 MapReduce 作业和其他批处理作业使用。使用准入控制来控制集群中 Impala 作业对并发和内存的消耗，而使用 YARN 来控制 Hadoop 其他组件的资源消耗。

我们可以将 YARN、Impala、Llama 一起使用，其中 YARN 管理所有的集群资源，以 Llama 作为媒介所有的 Imapla 查询也从 YARN 中请求资源。YARN 是一个集中管理的更加通用的资源管理服务。Impala 通过 YARN 申请资源将要比准入控制具有更高的延迟，因为它必须通过 Llama 向 YARN 提交并返回请求。

Impala 准入控制提供了类似于 YARN 的机制，将用户映射到不同的池并提供必要的身份验证。虽然 YARN 只支持 CDH5 或者更高版本，但是 Cloudera 对 CDH4 的底层架构进行了修改，所以在 CDH4 上也是可用的。如果我们使用准入控制，那么也可以不运行 YARN 和 Llama。

在 Cloudera Manager 中，如果我们没有启用 Llama 角色，Impala 将使用准入控制单独对资源进行管理。如果启用了 Llama 角色，YARN 将通过 Llama 来控制 Impala 的资源管理。单独使用的准入控制功能为我们提供了三个属性：内存限制、查询队列大小、超时队列。如果启用了 Llama，那么我们可以修改的属性只剩下了查询队列大小和超时队列，而内存限制参数将由 YARN 通过动态资源池进行分配。

6.1.3 并发查询限制

准入控制机制被内嵌在每个 impalad 节点进程中，通过 statestore 进程进行相互通信。虽然我们在集群侧设置了内存使用限制和并发查询的数目，但是对新的查询请求执行或者进入等待队列是由 impalad 进程自己决定的。这也就意味着准入控制的开销很低，但是在系统负载严重的时候可能不是非常精确。也就是说集群使用的内存和并发查询的个数可能会偶尔超过我们指定的限制。所以在设置内存限制时，我们可以设置一个略小于我们预期的值；而在设置等待队列大小时，我们可以设置一个略大于我们预期的值。

在查询等待队列中的查询包括集群中从任何 Impalad 进程节点提交的查询。所有通过特定节点提交的查询将会顺序被执行，比如，只有在执行了 CREATE TABLE 语句之后，向表中 INSERT 语句才会执行成功。而通过不同节点提交的查询将不会强制按顺序执行。基于这个原因，如果我们使用了负载均衡机制或者 round-robin 调度机制，这时，所有的语句都是通过不同的节点提交的，

而通过不同节点提交的语句我们没有办法保证其顺序，为了避免类似于表在没有创建之前就进行插入的错误，我们需要提前创建好表结构。也或者我们将 CREATE TABLE、INSERT 作为一个单独的执行单元，让这个执行单元可以在某一个特定的 impalad 进程节点上提交，以保证其执行顺序。

并发查询的数目也是一个软的限制。为了实现高吞吐量，Impala 在节点级别对传入的查询请求的那一时刻立即作出是执行还是放入等待队列的判断。而做出判断的那个时刻存在的偶然性也可能导致 Impala 可能时不时地略微超过限制的数目。

为了防止等待队列中的查询被大量积压，我们也可以为等待队列设置一个上限值。当等待队列中的查询请求达到上限值后，后续的查询将直接被终止而不是被放入等待队列。另外，我们可以设置一个超时时间，在超过超时时间后的所有等待队列中的查询会被终止，而不是无限期等待。如果一个集群经常出现超时，有过多的并发查询请求，或者查询执行时间过长的情况，可能就需要管理员介入了。管理员通过调查判断是应该添加更多的硬件资源，还是改善作业的调度，还是对系统进行性能优化。

6.1.4　准入控制和 Impala 客户端协同工作

准备控制对 JDBC ODBC 这样的客户端接口是透明的：

（1）如果 SQL 语句被放入等待队列，而不是立即执行，API 调用会一直阻塞到语句从等待队列出现并开始执行的时候。也就是说客户端应用可能立即接收到返回的结果，也可能一直阻塞等待返回结果。

（2）如果 SQL 语句由于超出队列长度限制或者内存限制被终止执行，客户端应用程序将会接收到一条描述性错误信息。

在使用了 JDBC 或者 ODBC 的应用程序中，准入控制有如下限制和特殊的行为：

（1）如果我们想通过 REQUEST_POOL 参数将查询提交到不同的资源池，REQUEST_POOL 参数必须在 impala-shell 的会话级别设置，或者在集群侧启动 impalad 进程时设置。

（2）使用 MEM_LIMIT 参数有时可以解决 Impala 对复杂查询的内存消耗估算不准确的问题，但是这个参数只能通过 impala-shell 来设置，无法通过 JDBC 或者 ODBC 直接设置。

（3）准入控制不能使用像 RESERVATION_REQUEST_TIMEOUT 或者 V_CPU_CORES 等资源相关的参数。这些参数只对基于 YARN 管理的资源管理框架有效。

6.1.5　配置准入控制

准入控制的基本配置很简单，比如我们可以只配置一个资源池和一组参数集合。如果要做到精细化的控制，配置可能要稍微复杂一些，比如我们要使用不同配置参数的多个资源池，每个资源池捕获特定用户或者组的查询请求。要完成这些配置，我们可以直接使用 Cloudera Manager 进行配置，也可以直接编辑配置文件或者控制 impalad 进程的启动参数来进行配置。

1. 使用 Cloudera Manager 配置

在 Cloudera Manager 管理控制台上，我们可以配置资源池，管理等待队列，设置并发查询的个数限制以及如何捕获到是否超过了限制等。

之后的示例中将显示如何使用 Cloudera Manager 来配置这些参数。

2. 手动配置

如果我们没有使用 Cloudera Manager，我们也可以通过修改配置文件 fair-scheduler.xml 和 llama-site.xml，并修改 impalad 进程的启动参数来实现。

对于一个只使用单个资源池（默认名称为 default）的简单配置，我们可以不配置 fair-scheduler.xml 和 llama-site.xml，只配置命令行参数。

impalad 进程与准入控制相关的参数说明如下：

（1）--default_pool_max_queued

等待队列允许的最大请求的数目。因为这个限制是集群侧的，每个 Impala 节点都可以独自决定传入的查询请求是运行还是放入等待队列，所以这也是一个软限制。在集群负载比较高时，等待队列中的查询请求可能会略高于限定的数目。如果将该参数指定为负数或者 0，表示一旦达到的并发请求的查询总数，后续的查询将会被拒绝。如果设定了 fair_scheduler_config_path 和 llama_site_path，将忽略该参数。

类型：int64。

默认值：0。

（2）--default_pool_max_requests

该参数表示 Impala 允许的并行执行的查询的最大个数。该参数与 default_pool_max_queued 一样，是一个软限制，实际生产环境中的并发执行的查询个数可能会偶尔略高于这个值。如果该参数为负数表示没有限制。如果设定了 fair_scheduler_config_path 和 llama_site_path，将忽略该参数。

类型：int64。

默认值：-1。

（3）--default_pool_mem_limit

该参数表示所有并发执行的查询可以消耗的内存的最大值。该参数支持的单位包括 B，M，G。对于 M 和 G，我们可以使用浮点数值，比如指定为 1.5M。如果不使用单位，默认的单位为字节，也就是 B。另外，我们还可以把该参数指定为物理内存的百分比。该参数是一个软性的限制。

类型：字符串。

默认值：“”空字符串（表示无限制）。

（4）--disable_admission_control

该参数表示关闭准入控制功能。

类型：BOOLEAN。

默认值：FALSE。

（5）--disable_pool_max_requests

该参数表示禁用所有的资源池对并发查询请求个数的限制。

类型：BOOLEAN。

默认值：FALSE。

(6)--disable_pool_mem_limits

该参数禁用所有资源池对内存的限制。

类型：BOOLEAN。

默认值：FALSE。

（7）--fair_scheduler_allocation_path

该参数指定公平调度器的配置文件 fair-scheduler.xml 的路径。准入控制只有一小部分配置可以在这个配置文件中配置。

类型：STRING。

默认值：“”空字符串。

（8）--llama_site_path

该参数指定 Llama 配置文件 llama-site.xml 的路径。如果设置了该参数，fair_scheduler_allocation_path 参数也必须被设置。准入控制只有一小部分配置可以在这个配置文件中配置。

类型：STRING。

默认值：“”空字符串。

（9）--queue_wait_timeout_ms

该参数表示一个请求被接受之前需要等待的最长毫秒数。

类型：int64。

默认值：60000。

对于更高级的使用多资源池的配置，我们需要手动修改 fair-scheduler.xml 和 llama-site.xml 配置文件。同时在 Impalad 进程的命令行参数中配置 --fair_scheduler_allocation_path 和 --llama_site_path 选项指定参数文件的位置。

Impala 准入控制功能只使用完全公平的调度器配置用户或者组和不同资源池的映射关系。比

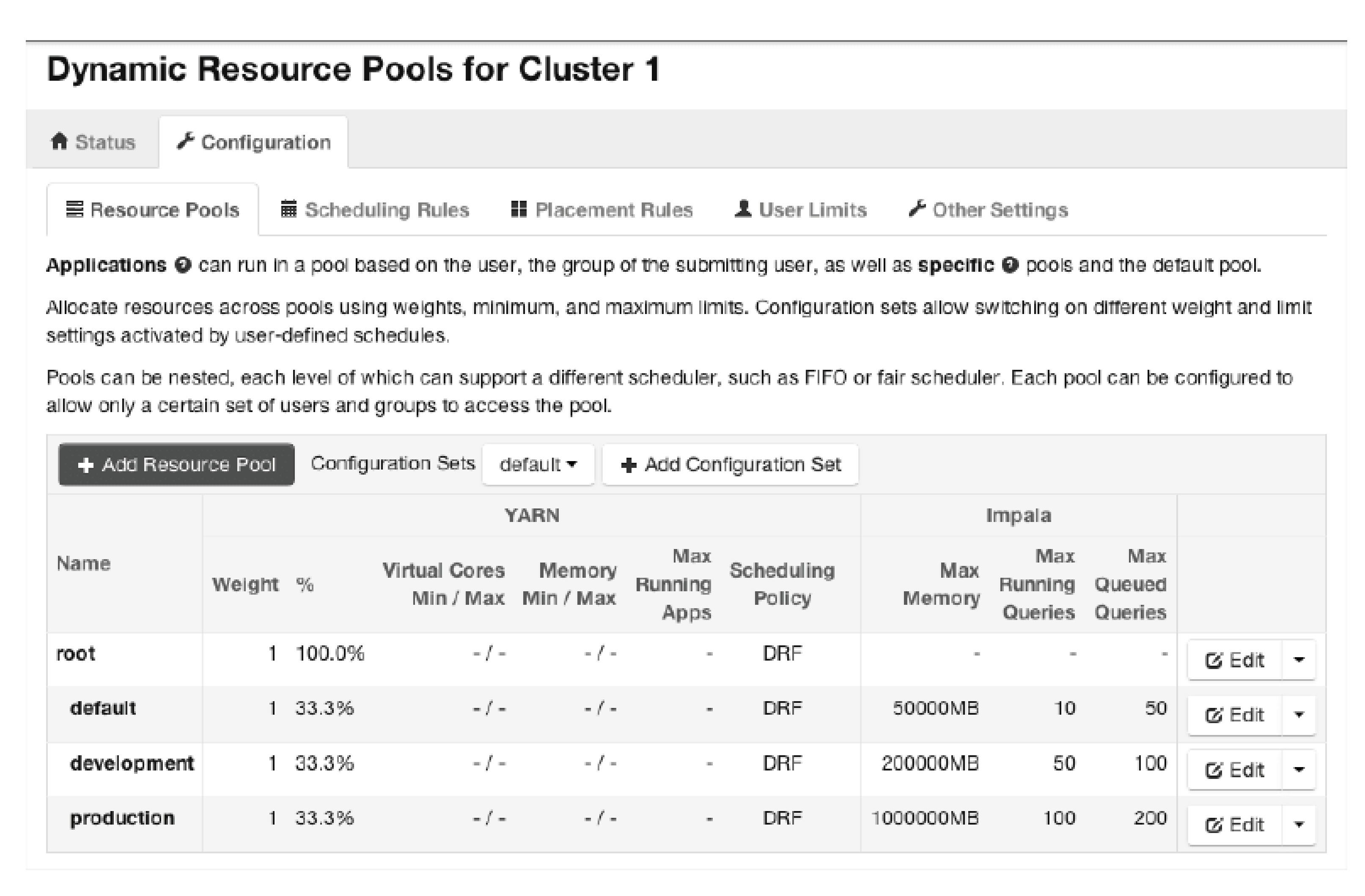
如，我们可以使用不同的内存限制、并发查询数、等待队列的长度的资源池以满足不同用户的需要。

Impala 准入控制使用的 Llama 配置文件中的属性包括：

```
llama.am.throttling.maximum.placed.reservations.queue_name
llama.am.throttling.maximum.queued.reservations.queue_name
```

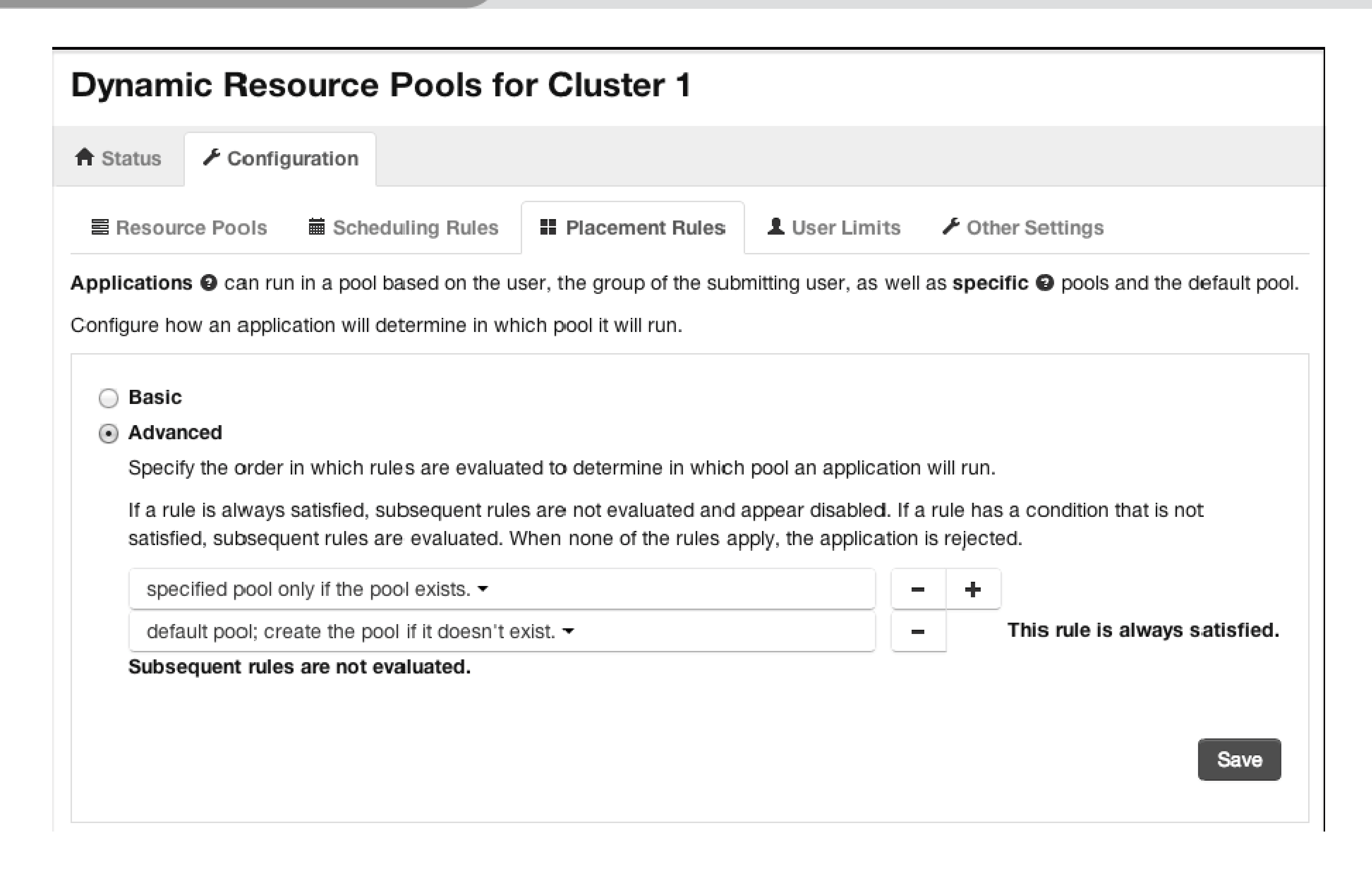
3. 准入控制配置示例

下图显示了 Cloudera Manager 中动态资源池页面的示例。root 池包括所有的资源池。default 池用来指向那些没有显式的分配资源池的用户。development 池和 production 池则演示了我们可以为不同类的用户指定不同的限制。



下图的预置的规则配置了一个名称为 default 的资源池接收如下情况的请求：

- 指定资源池不存在
- 未显示指定资源池
- 用户或组没有指定资源池



对于不使用 Cloudera Manager 管理的集群，我们提供了与上述资源池定义类似的 fair-scheduler.xml 和 llama-site.xml 样例配置文件。这个示例文件是从生产环境中取下来的，包括了 YARN 和 Llama 配置的不同方面。

```
fair-scheduler.xml
```

即使 Impala 不使用 vcores 值，我们也必须匹配 YARN 的要求制定该值。每个<aclSubmitApps>标签包含了这样的一个格式：一个逗号分隔的用户列表，一个空格，一个逗号分隔的组列表。这里制定的用户和组可以访问与这个标签对应的资源池。如果这对标签中间是空的，那么表示所有人都不能直接使用这个资源池。子资源池可以拥有自己的<aclSubmitApps>标签指定可以访问自己的用户和组。

```
<allocations>
<queue name="root">
<aclSubmitApps> </aclSubmitApps>
<queue name="default">
<maxResources>50000 mb, 0 vcores</maxResources>
<aclSubmitApps>*</aclSubmitApps>
</queue>
<queue name="development">
<maxResources>200000 mb, 0 vcores</maxResources>
<aclSubmitApps>user1,user2 dev,ops,admin</aclSubmitApps>
</queue>
<queue name="production">
<maxResources>1000000 mb, 0 vcores</maxResources>
<aclSubmitApps> ops,admin</aclSubmitApps>
```

```
</queue>
</queue>
<queuePlacementPolicy>
<rule name="specified" create="false"/>
<rule name="default" />
</queuePlacementPolicy>
</allocations>


llama-site.xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<property>
<name>llama.am.throttling.maximum.placed.reservations.root.default</name>
<value>10</value>
</property>
<property>
<name>llama.am.throttling.maximum.queued.reservations.root.default</name>
<value>50</value>
</property>
<property>
<name>llama.am.throttling.maximum.placed.reservations.root.development</name>
<value>50</value>
</property>
<property>
<name>llama.am.throttling.maximum.queued.reservations.root.development</name>
<value>100</value>
</property>
<property>
<name>llama.am.throttling.maximum.placed.reservations.root.production</name>
<value>100</value>
</property>
<property>
<name>llama.am.throttling.maximum.queued.reservations.root.production</name>
<value>200</value>
</property>
</configuration>
```

6.1.6 使用准入控制指导原则

为了确认准入控制是否对查询生效，我们可以检查查询的 PROFILE 的输出。在 impala-shell 中执行完一个查询之后立刻运行 PROFILE，在 Impala 调试的 web 界面或者在日志文件中，我们就可以看到相关信息。这些信息包括像查询是否进入了等待队列，使用了哪个资源池等准入控制的相关信息。它还包括了估算的内存消耗和真实的内存消耗，通过这些信息我们可以很好的调整资源池的内存限制参数。

在实践中，我们推荐使用 Cloudera Manager 来配置准入控制的相关参数。毕竟通过 GUI 来修改配置信息要比手动修改配置文件要简单的多。而在 Cloudera Manager 4 中，我们不能通过图形界面来进行配置，只能使用为 impalad 进程指定启动参数的方式。

我们要时刻谨记所有的准入控制的参数都是软限制。所有的是否超出限制的判断都是 Impalad 进程节点依赖于 statestore 服务传送的信息单独作出的判断。瞬时骤增的查询请求可能导致内存的使用超出 Impala 的软限制，作为后备方案，我们可以使用 Linux 的 cgroups 机制来硬性限制内存的使用，以防止内存过载的情况出现。

如果我们遇到了一个查询由于 Impala 对它的估算内存过高而带来的问题，我们可以在 impala-shell 中设置 MEM_LIMIT 参数覆盖估算内存值，然后在同一个会话中执行该查询。这个值会被视为 Impala 对查询的估算值覆盖 Impala 真正通过表和列统计信息计算出来的估算值。另外，这个值仅用于准入控制的判断，而不是真的要给查询预分配这么多的内存。

在 impala-shell 中，我们可以直接设置 REQUEST_POOL 参数用于指定使用哪个资源池。

被准入控制影响的语句主要是查询语句，但是也包括像写数据的 INSERT 或者 CREATE TABLE SELECT 语句等。Impala 中大多数写操作消耗的资源都不很明显，但是如果是想 Parquet 表中插入数据，Impala 要为每个 1GB 的 Parquet 数据块消耗 1GB 的内存作为缓存，这是不小的内存消耗。

虽然准入控制不检查 DDL 语句对内存的消耗情况，但是如果碰巧达到了并发数或者内存限制，DDL 语句也会进入等待队列。如果系统有了足够的资源，通过同一个会话进入队列的语句会依据原有的顺序进行执行：

```
select * from huge_table join enormous_table using (id);
```

对于大表关联这样高内存消耗的语句，在 Impala 计算（判断）它如果运行将导致现有集群的内存超出限制时，该语句将进入等待队列。

```
drop table huge_table;
```

如果第一个语句进入等待队列，那么后续的语句，哪怕是 DDL 语句也会进入等待队列。

如果我们为不同的用户和组设置了不同的资源池，我们可以考虑重用在 Sentry 中使用的分类和层次。

6.2 使用 YARN 资源管理(CDH5)

在 Impala 中，我们可以通过限制 CPU 和内存资源来控制集群的负载。在 CDH5 中，Impala 也可以运行在基于 YARN 的资源管理的框架之下，由 YARN 负责为每个 Impala 查询分配资源。Impala 负责计算每个查询需要的资源，并向 YARN 提交申请。

通过 Impala 向 YARN 申请资源的请求使用过 Llama 服务调用进行的。如果 Impala 的资源请求被允许，那么它将所有的执行线程放入 Cgroup 容器，为每个节点设置内存限制并启动查询。如果此时集群没有足够的资源可用，Impala 将会持续等待，直到其他的作业执行结束有了足够的资源为止。

在查询执行完成之后，Llama 将缓存已经使用的资源（比如，内存不会被释放），这样就不必为后续的 Impala 查询频繁创建这些资源。如果是 YARN 中其他类型的作业需要这些资源，Llama 就会将这些资源返还给 YARN。

在一个负载压力比较大的集群上，某些查询由于对资源的延迟等待可能响应时间会变的无法控制，然而这样做的优势是整个集群的性能负载变化会比较均匀，不会出现那种 CPU 突然利用率达到百分之百或者内存突然不够用而导致换页的情况。

6.2.1　Llama 进程

Llama 是位于 Cloudera Impala 和 Hadoop YARN 之间的资源管理系统。Llama 可以控制 Impala 预留、使用、释放从 Hadoop 集群申请的资源。如果要使用 YARN 对 Impala 进行资源管理，Llama 是必须的。

默认情况下，对于 MapReduce 作业，YARN 采用按位分配资源的方式。而对于 Impala 作业，为了保证中间的计算结果在集群各节点之间交换，而不是让查询在执行过程中再去等待申请资源，它需要所有的资源必须同时申请到，同时可用。Llama 可以确保在 Impala 查询开始执行之前所有的集群中所有节点的资源同时申请到位。

6.2.2　检查计算的资源和实际使用的资源

为了使资源使用情况便于确认，SQL 的 EXPLAIN 命令输出的执行计划包括了关于内存估算情况，表和列统计信息是否可用，查询将要使用的虚拟核数等信息。我们可以不运行查询，只对语句执行 EXPLAIN 来查看这些信息。如果想了解更多的详细的信息，需要指定查询参数 EXPLAIN_LEVEL=verbose。类似的，在 impala-shell 中使用 PROFILE 语句也可以看到这些相关的信息。但是如果想通过 PROFILE 看到详细的信息，需要先真正地运行过（查询）一次才可以。如果我们发现在启用了资源管理的系统上查询语句运行失败或者有性能问题，我们首先需要做的就是搜集统计信息，调整查询参数。

6.2.3 资源限制如何生效

CPU 的限制通过 Linux 的 Cgroup 机制实现。YARN 为与各机器关联的 Cgroup 组的容器分配授权资源。

内存限制通过 Impala 的查询内存上线进行限制。一旦一个查询请求被授权允许执行，Imapla 在查询执行之前就会根据所授予的内存数量对查询能够使用的内存进行限制。

6.2.4 启用 Impala 资源管理

如果要启用 Impala 的资源管理，我们必须首先在集群中安装 YARN 和 Llama 服务。

YARN 是一个对 Hadoop 集群组件进行资源管理的通用服务，Llama 是作为 YARN 和 Impala 之间资源管理的桥接服务：将 Impala 的查询请求发送到 YARN，确保 Impala 在查询在执行之前能够从 YARN 申请到所有可用的资源。

接下来介绍一下 Impala 资源管理启动参数。以下是启用 Impala 资源管理后的集群定制化参数：

- -enable_rm：是否启用资源管理。这是一个布尔型参数，默认值为 FALSE。如果该参数为默认值，那么其他的资源管理的参数都不会生效。
- -llama_host：Impala 需要连接到的 Llama 服务的主机名或者 IP 地址。默认值为 127.0.0.1。
- -llama_port：Impala 需要连接到的 Llama 服务的端口号。默认值为 15000。
- -cgroup_hierarchy_path：YARN 和 Llama 需要创建的用于授权资源的 Cgroup 组的路径。Impala 默认使用的容器分配的 Cgroup 组的路径为"cgroup_hierarchy_path +container_id"。

6.2.5 资源管理相关 impala-shell 参数

在通过 impala-shell 执行 SQL 之前，我们可以通过 SET 命令设定资源管理的相关参数有：

- EXPLAIN_LEVEL
- MEM_LIMIT
- RESERVATION_REQUEST_TIMEOUT
- V_CPU_CORES

这些参数在 impala-shell 章节中已介绍。

6.2.6 Impala 资源管理的限制

目前，在 CDH5 中，对 Impala 查询有如下限制：

- 表和列统计信息对于 Impala 估算这个查询请求需要的内存量是一个非常重要的参考依据

- 当 Impala 为一个查询实际申请的内存超过它估算的内存时，这个查询将会终止执行。即使表和列统计信息是准确的，对于某些复杂查询 Impala 也可能估算出现偏差导致其终止执行。在一个查询执行失败之后，我们可以在 impala-shell 中执行 PROFILE 命令来确认它实际使用的内存数量。我们可以将 MEM_LIMIT 设定为一个更大的值重新执行查询。
- MEM_LIMIT 和其他的查询参数一样，无法通过 JDBC 或者 ODBC 接口来控制。

6.3 为进程，查询，会话设定超时限制

我们可以依据 CDH 集群的繁忙程度来增减不同的超时参数。

1. 增加 Statestore 超时参数

如果我们的 Impala 包含太多的元数据（比如有上千个数据库，上万张表的集群），在集群启动时需要使用 Statestore 服务来将元数据广播到所有节点时可能出现超时错误。为了避免这种超时错误，我们需要将集群默认的 10 秒钟的超时时间改大。我们可以通过 -statestore_subscriber_timeout_seconds 参数来设定。在出现这种超时错误是，impalad 日志可能出现如下信息：

```
Connection with state-store lost
Trying to re-register with state-store
```

2. 为 impalad 设定空闲超时

为了防止长时间运行的查询或者空闲的会话对集群资源的浪费，我们可以为单独的查询或者整个会话设置超时时间。在 impalad 进程启动时，我们需要设置以下参数：

- --idle_query_timeout：在超过了指定的时间后，空闲的查询将会被终止。这里的空闲查询指的是一个已经返回了所有结果集但是没有关闭的查询，或者是已经返回了部分结果但是客户端终止了查询请求的查询。这些情况不会出现在 impala-shell 中，但可能经常出现在 JDBC 或者 ODBC 接口调用时。一旦这个查询被终止，客户端将无法获取结果。
- --idle_session_timeout: 该参数用于指定空闲会话的超时时间。这里的空闲会话指的是一个没有运行任何查询的会话。一旦会话超时，虽然它仍然处于打开状态，但是我们将不能通过这个会话发送任何查询请求，我们唯一可以做的动作就是关闭这个会话。该参数默认值为 0，表示会话永远不会超时。

6.4 通过代理实现 Impala 高可用性

对于一个繁忙的，有很大负载的一个集群来说，我们可以需要配置一个代理服务器来转发 Impala 的请求。这样做的优势如下：

- 应用程序只需要连接到单台机器的一个端口，而无须关注具体的 impalad 进程节点信息。
- 如果某个 impalad 进程节点失效，客户端应用程序仍然可以正常向代理服务器转发请求。
- 协调者节点一般情况下会消耗比其他节点更多的 CPU 和内存资源。代理服务器可以利用 ROUND-ROBIN 算法将查询请求发送到不同的节点，因此每次连接都使用不同的协调者节点，避免了单个节点的资源过度消耗。

如果要使用负载均衡技术，我们可以大致按照以下步骤进行：

步骤 01 下载负载均衡代理服务器软件，并进行安装。

步骤 02 对软件进行配置，设置用来转发 Impala 请求的相关配置信息。

步骤 03 指定每个 Impala 节点的主机名称和端口号。代理服务器将查询请求转发到我们指定的主机端口。

步骤 04 使用相应的配置文件来启动代理服务器软件。

1. 使用 Kerberos

如果集群启用了 Kerberos，为了避免人为的攻击，应用程序会检查主机凭据以确认正在连接到的主机和真正处理请求的主机是同一台。为了让 Kerberos 确认我们的连接请求的合法性，我们需要执行以下步骤：

步骤 01 确认集群启用了 Kerberos。

步骤 02 确定我们将要使用的代理服务器的主机。使用基本的 Kerberos 安装，在它的 keytab 中应该已经有了类似与 impala/proxy_host@realm 的条目。如果没有该条目，我们需要重新初始化 Kerberos 配置过程。

步骤 03 将 keytab 文件拷贝到所有的 Impalad 进程节点上。将该文件放在每台主机的相对安全的位置上。

步骤 04 为每台 Impalad 进程节点的 keytab 添加条目 impala/actual_hostname@realm。

步骤 05 在每个 impalad 进程节点上，使用 ktutil 工具合并已存在的 keytab 和代理服务器的 keytab，生成一个新的 keytab 文件。

```
$ ktutil
ktutil: read_kt proxy.keytab
ktutil: read_kt impala.keytab
ktutil: write_kt proxy_impala.keytab
```

```
ktutil: quit
```

步骤 06　确保 impala 用户对该文件有读权限。

步骤 07　修改每个参与负载均衡的 impalad 进程节点的启动参数。在 Impalad 参数定义或者 Cloudera Manager 安全阀字段添加：

```
--principal=impala/proxy_host@realm
--be_principal=impala/actual_host@realm
--keytab_file=path_to_merged_keytab
```

步骤 08　如果使用 Cloudera Manager 管理，则需要重启 Impala 服务。

如果没有使用 Cloudera Manger 管理，需要重启所有 Impalad 进程节点和 statestored、catalogd 服务。

2. HAProxy 配置示例

如果我们还没有使用负载均衡代理服务器，我们可以选择 HAProxy。它是一个免费、开源的负载均衡器。如下示例我们将介绍如何在 RedHat Linux 企业版上安装和配置该负载均衡器。

步骤 01　安装负载均衡器。

```
yum install haproxy
```

步骤 02　设置配置文件 haproxy.cfg。见稍后示例。

步骤 03　运行负载均衡器。（负载均衡代理服务器最好不要使用 Impalad 进程节点。）

```
/usr/sbin/haproxy -f /etc/haproxy/haproxy.cfg
```

步骤 04　在 impala-shell，JDBC 或者 ODBC 应用程序中不要再连接到 Impalad 进程节点，而是连接到代理服务器 haproxy_host:25003。

3. 配置文件 haproxy.cfg 示例

```
global
# To have these messages end up in /var/log/haproxy.log you will
# need to:
#
# 1) configure syslog to accept network log events. This is done
# by adding the '-r' option to the SYSLOGD_OPTIONS in
# /etc/sysconfig/syslog
#
# 2) configure local2 events to go to the /var/log/haproxy.log
# file. A line like the following can be added to
```

```
# /etc/sysconfig/syslog
#
# local2.* /var/log/haproxy.log
#
log 127.0.0.1 local0
log 127.0.0.1 local1 notice
chroot /var/lib/haproxy
pidfile /var/run/haproxy.pid
maxconn 4000
user haproxy
group haproxy
daemon
# turn on stats unix socket
#stats socket /var/lib/haproxy/stats
#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#
# You might need to adjust timing values to prevent timeouts.
#---------------------------------------------------------------------
defaults
mode http
log global
option httplog
option dontlognull
option http-server-close
option forwardfor except 127.0.0.0/8
option redispatch
retries 3
maxconn 3000
contimeout 5000
clitimeout 50000
srvtimeout 50000
#
# This sets up the admin page for HA Proxy at port 25002.
#
listen stats :25002
balance
mode http
```

```
    stats enable
    stats auth username:password
    # This is the setup for Impala. Impala client connect to load_balancer_host:25003.
    # HAProxy will balance connections among the list of servers listed below.
    # The list of Impalad is listening at port 21000 for beeswax (impala-shell) or
original
    ODBC driver.
    # For JDBC or ODBC version 2.x driver, use port 21050 instead of 21000.
    listen impala :25003
    mode tcp
    option tcplog
    balance leastconn
    server symbolic_name_1 impala-host-1.example.com:21000
    server symbolic_name_2 impala-host-2.example.com:21000
    server symbolic_name_3 impala-host-3.example.com:21000
    server symbolic_name_4 impala-host-4.example.com:21000
```

6.5 管理磁盘空间

虽然 Impala 存储通常使用 HDFS 上的许多大文件，但是我们还是可以通过文件清理释放空间，或者让开发者减少文件复制等尽可能少的消耗空间。

（1）在生产环境中使用二进制压缩文件。其中数字类型和时间日期类型压缩率较高。但是不同的文件格式提供的压缩比不同。我们可以在 CREATE TABLE 时使用 STORED AS 选项或者使用 ALTER TABLE 时指定 SET FILEFORMAT 子句来改变表或者分区的存储格式。

（2）我们可以管理 Impala 内部表和外部表的数据文件。

使用 DESCRIBE FORMATTED 语句来确认表是内部表还是外部表，在 HDFS 上的具体位置信息等。

对于 Impala 内部表，我们可以使用 DROP TABLE 命令删除数据文件。

对于 Impala 外部表，我们可以使用像 hadoop fs、hdfs dfs、distcp 等 Hadoop 相关的命令来创建、移动、拷贝或者删除文件。对 HDFS 上的文件操作执行完成之后，我们需要执行 REFRSH table_name 命令更新外部表的元数据信息。

使用直接指向 HDFS 上原始数据文件的位置信息创建 Impala 的外部表可以有效的避免对数据文件的复制。我们可以把一个数据文件映射到多张 Impala 外部表。当我们删除 Impala 外部表时，数据文件不会被级联删除。

使用 LOAD DATA 命令将 HDFS 上的文件移动到 Impala 的数据目录时，无须指定要加载的目

标位置信息。这项技术对于 Impala 的内部表和外部表都适用。

（3）确保 HDFS 的回收站信息正确配置。当我们从 HDFS 上删除文件之后，相应的空间可能由于回收站的配置而不被释放。

（4）在删除一个数据库之前请先删掉数据库中所有的表。

（5）在 INSERT 语句执行失败后及时清理临时文件。如果一个 INSERT 语句执行失败，我们可以在数据目录中看到一个名为.impala_insert_staging 的目录，这个目录包含插入数据的临时文件，会占用空间。如果是由于权限错误等其他外部原因而导致无法移动这些文件，那么我们可以根据提示纠正错误。如果是我们是由于内部错误无法插入数据，那么我们可能需要使用 Hadoop 文件系统的命令删除这些文件。使用 DESCRIBE FORMATTED table_name 语句可以查看临时文件在 HDFS 上的位置信息。

第 7 章

Impala存储

Impala 支持 Apache Hadoop 常用的文件格式。Impala 可以加载或者查询通过像 Pig 或者 MapReduce 等其他组件产生的数据文件，同样的 Impala 产生的数据文件也可以为其他 Hadoop 组件所使用。本章将讨论每种文件格式的使用、限制，以及对性能的影响。

对于同样数据的 Impala 表，使用不同的文件格式存储对性能的影响非常大。某些文件格式支持压缩功能，这将直接影响存储的文件的大小，读取该表产生的 IO，以及用于解压缩数据的 CPU 消耗等。通常情况下，我们将数据存储为压缩格式的。通过压缩数据，我们在将数据读入内存时会消耗更少的 IO 和内存资源，但同时由于多了解压数据的过程，会消耗更多的 CPU 资源。

7.1 文件格式选择

Impala 支持 Hadoop 中流行的大多数文件格式和压缩方式。有些文件格式，Impala 可以直接创建该格式文件的表，并对其进行查询操作；而对于另外一些文件格式，Impala 不支持直接写入操作，我们需要通过 Hive 创建表，并通过 INVALIDATE METADATA 语句将元数据信息同步到 Impala，之后就可以通过 Impala 来对数据进行查询了。文件格式也可以被结构化，在这种情况下它自身包含元数据信息或者内置的压缩类型。支持的文件格式如下表所示：

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
Parquet	结构化	Snappy，GZIP 当前默认 Snappy	是	是。CREATE TABLE、INSERT、LOAD DATA 或者查询均可
Text	非结构化	LZO	是。使用 CREATE TABLE 创建表是默认的文件格式，各字段通过 ASCII 码 0x01 分隔（也就是 Ctrl-A）	是。CREATE TABLE、INSERT、LOAD DATA 或者查询均可。如果使用了 LZO 方式压缩，必须在 Hive 完成创建表和加载数据的过程

（续表）

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
Avro	结构化	Snappy，GZIP，deflate，BZIP2	是。在 Impala 1.4.0 或者更高版本可以，但是在低版本中必须通过 Hive 创建	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载
RCFile	结构化	Snappy，GZIP，deflate，BZIP2	是	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载
SequenceFile	结构化	Snappy，GZIP，deflate，BZIP2	是	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载
HFile（HBase）	结构化	GZIP，LZO，Snappy	否。必须通过 Hive 创建	是。支持 INSERT 操作

Impala 支持如下的压缩编码：

- Snappy。这是推荐使用的压缩算法，因为它在压缩率和解压速度上实现了很好的平衡。Snappy 的速度非常快，但是 GZIP 更节省空间。该算法不支持文本文件。
- GZIP。当我们以节省磁盘空间为目的，需要更高的压缩比时，这种算法提供了一个很好的选择。该算法不支持文本文件。
- Deflate。该算法不支持文本文件。
- BZIP2。该算法不支持文本文件。
- LZO。该算法只支持文本文件。Impala 可以直接查询基于 LZO 压缩的表，但是目前不能直接创建基于 LZO 压缩的表，也不能直接向其中插入数据，创建和插入数据的操作必须通过 Hive 来完成。

不同的文件格式和压缩方式与具有不同特点的数据集能够更好的协同工作。无论使用哪种文件格式，Impala 都可以在一定程度上提供良好的性能，但是为具有不同特点的数据选择合适的文件格式，我们能够获得更大的性能提升。对于一个指定的表，我们应该使用哪种文件格式，哪种压缩方式可以依据以下几点：

（1）如果已经存在的数据文件格式是被 Impala 支持的，那么我们的 Impala 表通常使用同样的文件格式。除非原始的文件格式的性能、资源使用率等不能满足应用的要求，那我们就要使用不同的文件格式和压缩方式来重建这张表。重建时，我们可以使用 INSERT 语句将所有数据拷贝到新表中。当然，我们要依据新选择的文件格式是否被 Impala 支持来确定是在 Impala 还是在 Hive 中执行 INSERT 操作。

（2）通常情况下，很多工具可以直接生成文本文件，而且文本文件因为其易读性也决定了它

方便校验和调试。这些都是为什么 Impala 将其作为 CREATE TABLE 默认文件格式的原因。但是如果性能和系统资源利用率是我们最关注的要求时，我们需要选择其他的文件格式和压缩方式。很多时候，我们都是通过拷贝 CSV 或者 TSV 文件将数据加载到 Impala 表中，然后再使用 INSERT … SELECT 的方式将数据复制到基于其他数据格式的 Impala 表中。

（3）如果我们自身的架构将被查询的数据存放在内存中，那么请不要使用压缩。因为在内存中查询没有与磁盘交互的 IO 消耗，同时又增加了对数据解压时 CPU 的负载。

7.2 Text

Impala 支持使用 Text 文件作为存储的文件格式来。文本文件便于阅读，便于在不同的应用之间交换数据。

文本文件的列定义非常灵活。例如，一个文本文件可以包含比 Impala 表定义的更多的字段。在 Impala 进行查询时，将会忽略那些多出来的额外的字段。如果实际的文本文件包含的字段比表定义中的少，Impala 表中定义的比实际文件多的字段将统统被认为是 NULL 值。在 Impala 表中我们可以将文本文件中的字段作为数据类型或者时间戳类型来对待，也可以使用 ALTER TABLE … REPLACE COLUMNS 来进行转换。文本文件类型的特点如下表所示。

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
Text	非结构化	LZO	是。使用 CREATE TABLE 创建表是默认的文件格式，各字段通过 ASCII 码 0x01 分隔（也就是 Ctrl-A）	是。CREATE TABLE,INSERT,LOAD DATA 或者查询均可。如果使用了 LZO 方式压缩，必须在 Hive 完成创建表和加载数据的过程

7.2.1 查询性能

以文本格式作为存储文件格式占用的体积较大，而且不像 Parquet 这样的二进制文件格式效率高。如果我们接收到的数据就是文本格式的，而且我们不想做进一步处理，或者我们对其他的文件格式不够熟悉，而且使用文本格式就可以满足应用的需求情况下，我们会选择该格式作为数据存储。另一方面，如果应用对性能要求很高，我们必须尝试将文件格式替换为其他二进制文件格式。

对于频繁查询的数据，我们可以先将原始的文本数据文件加载到 Impala 表中，然后使用 INSERT 语句将存储的文件格式转换为其他的文件格式。如果目标表定义为特定的二进制文件格式，在数据插入的过程中转换会自动进行。

对于更紧凑的数据，我们可以考虑对文本文件进行 LZO 压缩。LZO 是 Impala 支持的对文本格式的唯一压缩方式。由于 LZO 数据文件天然的可分割特性，我们可以将同一个 LZO 压缩文件

的不同部分发送到不同节点上并行运算。

7.2.2 创建文本表

如果我们不清楚像字段分隔符之类的文本数据文件的准确格式，我们可以使用不带后续子句的 CREATE TABLE 语句创建表：

```
create table my_table(id int, s string, n int, t timestamp, b boolean);
```

默认情况下创建的文本格式的数据文件可以接收以 Ctrl-A 字符（0x01）作为分隔符插入的列值。

一种更通用的情况是将已存在的文本数据文件导入到 Impala 表。这时使用的语法更为详细，需要指定 FIELDS TERMINATED BY 子句，而且前面必须有对应的 ROW FORMAT DELIMITED 子句。该语句可以以 STORED AS TEXTFILE 作为结束，也可以不指定这个选项，因为文本格式是 Impala 默认的文件格式。例如：

```
create table csv(id int, s string, n int, t timestamp, b boolean)
row format delimited
fields terminated by ',';
create table tsv(id int, s string, n int, t timestamp, b boolean)
row format delimited
fields terminated by '\t';
create table pipe_separated(id int, s string, n int, t timestamp, b boolean)
row format delimited
fields terminated by '|'
stored as textfile;
```

如上示例展示了我们可以将像 CSV、TSV，或者以管道符分隔，这样具有特定分隔符的文本文件导入到 Impala 表中。我们也可以对这些表使用 INSERT … SELECT 语法将数据文件拷贝到 Impala 数据目录中。

在 Impala 1.3.1 或者更高版本中，我们可以指定'\0'字符作为分隔符：

```
create table nul_separated(id int, s string, n int, t timestamp, b boolean)
row format delimited
fields terminated by '\0'
stored as textfile;
```

执行 DESCRIBE FORMATTED table_name 语句可以查看在 Impala 底层的更多详细信息。

7.2.3　数据文件

当 Impala 对文本表进行查询时，它会遍历这个表对应的数据目录中的所有数据文件，但是会忽略以“.”开头的那些文件。

通过 Impala 的 INSERT 语句插入数据时，底层将自动为数据文件生成一个唯一名称以避免和其他文件名冲突。

一个 INSERT ... SELECT 语句会为在该节点上处理的 SELECT 的数据产生一个数据文件。而每条 INSERT ... VALUES 语句将产生一个单独的数据文件。Impala 在对少量的大数据文件查询的效率更高，所以强烈不建议使用 INSERT ... VALUES 的方式加载批量数据。如果我们已经由于使用了 INSERT ...VALUES 语句产生大量的小文件而导致的效率问题，我们必须通过 INSERT ... SELECT 的方式将数据迁移至另外一张表中。

文本数据文件特殊值说明如下：

（1）Impala 对于 FLOAT 和 DOUBLE 数据列使用字符串 inf 表示无穷，使用 nan 表示不是一个数字。

（2）Impala 将字符串“\N”表示为 NULL。当使用 Sqoop 时，需要指定--null-non-string 和--null-string 确保所有的 NULL 值可以被正确处理。默认情况下，Sqoop 使用字符串“null”表示为一个 NULL 值，这可能导致 Impala 行转换错误。对于已经存在的数据文件的表我们也可以通过设置表的属性解决这个问题：

```
ALTER TABLE name SET TBLPROPERTIES ("serialization. null. format"="null")
```

7.2.4　加载数据

为了将已经存在的文件文件加载到 Impala 文本表中，我们可以使用 LOAD DATA 语句指定数据文件在 HDFS 上的路径，也可以直接将文件拷贝到 Impala 数据目录中。

为了将已经存在的多个数据文件加载到 Impala 文本表中，我们也可以通过 LOAD DATA 语句指定这些文件所在的路径，或者将其直接拷贝到 Impala 数据目录中。

如果我们想把其他二进制格式的数据文件转换成文本，我们可以使用如下 SQL 语句：

创建一张与二进制格式类似表结构的文本表。

```
CREATE TABLE csv LIKE other_file_format_table;
```

指定文本表的字段分隔符。

```
ALTER    TABLE    csv    SET    SERDEPROPERTIES    ('serialization.format'=',',
'field.delim'=',');
```

将其他二进制格式表的数据以 INSERT 的方式插入文本表。

```
INSERT INTO csv SELECT * FROM other_file_format_table;
```

这项技术在我们想查看 Impala 如何表示文本格式数据文件中的特殊值时非常有用。通过 DESCRIBE FORMATTED 语句，我们可以查看数据文件存储的底层 HDFS 目录，然后使用像 hdfs dfs -ls hdfs_directory 或者 hdfs dfs -cat hdfs_file 可以查看 Impala 创建的文本文件的内容。

如果是为了测试的目的想创建示例表，可以使用 INSERT … VALUES 语句创建：

```
INSERT INTO text_table VALUES ('string_literal',100,hex('hello world'));
```

如果我们想手动创建一个 Impala 文本表使用的文本文件，需要使用 \N 来表示 NULL 值。

我们也可以手动的使用 hdfs dfs -put 或者 hdfs dfs -cp 命令将数据文件传到 Impala 表的数据目录中。当我们移动或者拷贝新的文件到 Impala 表的数据目录中后，必须执行 REFRESH table_name 命令更新表的元数据信息，让 Impala 可以识别到新的数据文件。

7.2.5 LZO 压缩

Impala 支持对文本格式的数据文件使用 LZO 压缩。Cloudera 推荐在生产环境中对文本文件进行 LZO 压缩。尤其是对 IO 密集型的查询，使用 LZO 压缩会大大减少从磁盘读取的数据量，但是同时由于要在内存中解压数据，会对 CPU 有额外的消耗。

Impala 支持 LZO 压缩的文本文件，但是不支持 GZIP 压缩的文本文件。LZO 格式压缩的文件可以透明的解压到不同的节点进行分布式处理，而 GZIP 格式压缩的文件不支持该特性，不适合于 Impala 这种分布式的处理方式。

目前，Impala 只能对 LZO 压缩的文本文件进行查询操作，如果要进行写操作，必须通过 Hive 使用 CREATE TABLE 并加载数据，然后在使用 Impala 进行查询。一旦我们创建了以及基于 LZO 压缩的文本表，我们就可以手动的将通过 lzop 或者其他类似命令产生的压缩文件直接放到 Impala 数据目录中。

1. 使用 LZO 准备工作

在 Impala 中使用 LZO 文本表之前，我们必须在集群的每个节点上安装必要的安装包。安装的步骤必须手动执行：

（1）安装前准备工作。

对于使用包管理的 Cloudera Manager 或者不被 Cloudera Manager 管理的系统，我们需要为自己的操作系统下载匹配的 GPL extras 的 repo 文件：

Red Hat 5 下载到 /etc/yum.repos.d/，下载地址为 http://archive.cloudera.com/gplextras/redhat/5/x86_64/gplextras/cloudera-gplextras4.repo。

Red Hat 6 下载到 /etc/yum.repos.d/，下载地址为 http://archive.cloudera.com/gplextras/redhat/6/x86_64/gplextras/cloudera-gplextras4.repo。

SUSE 下载到 /etc/zypp/repos.d/，下载地址为 http://archive.cloudera.com/gplextras/sles/11/x86_64/gplextras/cloudera-gplextras4.repo。

Ubuntu 10.04 下载到 /etc/apt/sources.list.d/，下载地址为 http://archive.cloudera.com/gplextras/

ubuntu/lucid/amd64/gplextras/cloudera.list。

Ubuntu 12.04 下载到 /etc/apt/sources.list.d/，下载地址为 http://archive.cloudera.com/gplextras/ubuntu/precise/amd64/gplextras/cloudera.list。

Debian 下载到 /etc/apt/sources.list.d/，下载地址为 http://archive.cloudera.com/gplextras/debian/squeeze/amd64/gplextras/cloudera.list。

（2）为 Impala 配置 LZO。

使用如下的安装命令，让 Hadoop 支持 LZO，让 Impala 支持 LZO。

```
RHEL/CentOS
$ sudo yum update
$ sudo yum install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo yum install hadoop-lzo # For clusters running CDH 5 or higher.
$ sudo yum install impala-lzo
SUSE
$ sudo apt-get update
$ sudo zypper install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo zypper install hadoop-lzo # For clusters running CDH 5 or higher.
$ sudo zypper install impala-lzo
Debian/Ubuntu
$ sudo zypper update
$ sudo apt-get install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo apt-get install hadoop-lzo # For clusters running CDH 5 or higher.
$ sudo apt-get install impala-lzo
```

（3）修改配置文件。

对于客户端和服务端的 core-site.xml 配置文件中，添加属性 com.hadoop.compression.lzo.LzopCodec：

```
<property>
<name>io.compression.codecs</name>
<value>org.apache.hadoop.io.compress.DefaultCodec,org.apache.hadoop.io.compres
s.GzipCodec,
org.apache.hadoop.io.compress.BZip2Codec,org.apache.hadoop.io.compress.Deflate
Codec,
org.apache.hadoop.io.compress.SnappyCodec,com.hadoop.compression.lzo.LzopCodec
</value>
</property>
```

（4）重启 MapReduce 服务和 Impala 服务。

2. 创建 LZO 文本表

一张 LZO 文本表必须使用如下的选项通过 Hive 创建：

```
STORED AS
INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
```

而且某些 Hive 的本地设置也需要修改，示例如下：

```
hive> SET mapreduce.output.fileoutputformat.compress=true;
hive> SET hive.exec.compress.output=true;
hive> SET
mapreduce.output.fileoutputformat.compress.codec=com.hadoop.compression.lzo.Lz
opCodec;
hive> CREATE TABLE lzo_t (s string) STORED AS
> INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'
> OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';
hive> INSERT INTO TABLE lzo_t SELECT col1, col2 FROM uncompressed_text_table;
```

一旦我们创建了 LZO 文本表，我们就可以在 Hive 中使用 INSERT… SELECT 语句将其转换为其他格式的表。

在通过 Hive 的 INSERT 插入数据之后，我们需要检查表对应的 HDFS 上的数据目录，确保数据文件具有.lzo 的扩展名。如果我们之前的某些配置不正确，可能导致存储的文件没有被正确的压缩，这时使用 Impala 访问这张表将报错，因为 Impala 认为这是一张压缩过的 LZO 文本表。

在将数据加载到 LZO 压缩表之后，我们必须为文件建立索引。建立索引的方式是使用 Linux 命令行运行一个 Java 类 com.hadoop.compression.lzo.DistributedLzoIndexer。这个 Java 类包含在 hadoop-lzo 包中。

下面运行示例：

```
$ hadoop jar /usr/lib/hadoop/lib/hadoop-lzo-cdh4-0.4.15-gplextras.jar
com.hadoop.compression.lzo.DistributedLzoIndexer /hdfs_location_of_table/
```

索引文件与数据文件的文件名相同，但是扩展名为.index。如果数据文件没有被索引，Impala 查询仍然可以执行，但是从远程 DataNode 读取数据的效率将非常非常低。

一旦我们创建了 LZO 压缩的文本表，将数据加载进来之后，对数据文件做了索引，我们就可以通过 Impala 对这张表进行查询了。对通过 Hive 创建的表第一次查询时，我们需要使用 INVALIDATE METADATA 命令更新元数据信息，让 Impala 能够识别到这张新的表。

7.3 Parquet

Impala 可以创建，管理，查询 Parquet 表。Parquet 是高效可扩展的基于列式存储的二进制文件格式。Parquet 表尤其擅长处理针对表中的某列或者某几列进行的扫描查询。例如：我们对一张包含很多列的宽表的但列进行像 SUM()、AVG()等聚集操作。每个数据文件包含某些行的集合，被称为行组。在一个数据文件中，存储是按列组织的，单列的值的数据类型相同也能以很高的压缩比进行压缩。对 Parquet 表基于某列的值进行的查询，都能使用最小的 IO 最快地返回结果。Parquet 文件类型的特点如下表所示：

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
Parquet	结构化	Snappy，GZIP 当前默认 Snappy	是	是。CREATE TABLE、INSERT、LOAD DATA 或者查询均可

7.3.1 创建 Parquet 表

如果要创建一张 Parquet 表，需要在 impala-shell 命令行中指定表名，列名，数据类型及存储的文件格式信息，示例如下：

```
[hadoop-cs1:21000] > create table parquet_table_name (x INT, y STRING) STORED AS PARQUET;
```

另外，我们也可以克隆另外一张 Parquet 表的列名和数据类型信息：

```
[hadoop-cs1:21000] > create table parquet_table_name LIKE other_table_name STORED AS PARQUET;
```

在 Impala 1.4.0 或者更高版本中，即使我们没有 Parquet 数据文件对应的表，单从数据文件本身就可以获得 Parquet 表的定义信息。如果我们有了表的定义信息，我们就可以依据这个表定义创建一张外部表指向 Parquet 数据文件，即可实现对 Parquet 数据文件的访问：

```
CREATE EXTERNAL TABLE ingest_existing_files LIKE PARQUET
'/user/etl/destination/datafile1.dat'
LOCATION '/user/etl/destination'
STORED AS PARQUET;
```

我们也可以使用 Parquet 数据文件的表定义信息创建一张空表，然后我们再使用 INSERT 或者 LOAD DATA 的方向向这张表中加载数据：

```
CREATE TABLE columns_from_data_file LIKE PARQUET
'/user/etl/destination/datafile1.dat'
```

```
    STORED AS PARQUET;
```

我们也可以创建基于 Parquet 的分区表：

```
    CREATE TABLE columns_from_data_file LIKE PARQUET
'/user/etl/destination/datafile1.dat'
    PARTITION (year INT, month TINYINT, day TINYINT)
    STORED AS PARQUET;
```

只要创建了一张 Parquet 表，我们就可以通过如下类似的命令插入数据了：

```
    [hadoop-cs1:21000] > insert overwrite table parquet_table_name select * from
    other_table_name;
```

上述示例中如果目标表和源表的列的数量或者列的名称不一致，我们就不能使用*，而必须显示的指定列名。

7.3.2 加载数据

使用什么样的技术向 Parquet 表中加载数据取决于源数据是存在 Impala 表中，还是存在 Impala 之外的裸数据文件中。

如果源数据存在 Impala 表中，只是表的文件格式不是 Parquet 的，我们只需要使用 INSERT … SELECT 语句将其转换成 Parquet 格式的表。在转换的过程中，我们还可以进行过滤、重新分区等操作。

当我们向分区表尤其是 Parquet 的分区表插入数据时，在 INSERT 语句中使用合适的 Hint 能够大大的降低操作对资源的使用，提升性能。

（1）只有在 Impala 1.2.2 或者更高版本中才能使用这些 Hint。

（2）我们可以在使用 INSERT 向 Parquet 分区表插入数据由于资源限制导致失败或者插入效率低下时可以使用这些 Hint。

（3）Hint 的关键字[SHUFFLE] [NOSHUFFLE]可以使用在 PARTITION 子句之后，SELECT 关键字之前。

（4） [SHUFFLE]这个 Hint 会尽可能的降低并发写入 HDFS 的文件的数目，并且为每个分区分配 1GB 的内存作为缓存使用。这降低大大减少 INSERT 操作的资源使用。在插入数据的过程中，由于同一分区的数据可能在不同的节点上进行插入操作，可能会引发节点间的数据传输。

（5）[NOSHUFFLE]将选择一个速度相对快，但是会产生很多小文件的执行计划。这样很可能引发超出系统固有的容量限制而导致 INSERT 操作执行失败。

（6）如果目标表包含源表的分区列，Impala 会自动选择使用[SHUFFLE]。如果源表没有列统计信息，则只有[NOSHUFFLE]可以生效。

（7）如果源表既包含分区列，而且又包含列统计信息，Impala 会根据列统计信息中列的非

重复值的个数选择使用[SHUFFLE]还是 [NOSHUFFLE]。也只有在这种情况下，我们可以使用 Hint 来强制让 Impala 选择我们希望的执行方式。

任何针对 Parquet 表的 INSERT 操作都要确保 HDFS 上有足够的空间可用。Parquet 数据文件默认的块大小为 1GB，如果空间不足，可能导致 INSERT 操作失败。

我们应该尽量避免使用 INSERT … VALUES 的方式向 Parquet 表插入数据，因为该语句会产生很多很多单独的小文件，毕竟 Parquet 的优势在于对 1GB 大小的块进行压缩，并行处理。

如果我们有其他组件产生的 Parquet 数据文件，我们可以通过以下方式快速的基于这些数据文件建立 Impala 表用于查询：

（1）通过 LOAD DATA 语句将数据文件移动到 Impala 表对应的数据目录。在移动的过程中，Impala 不会对文件进行校验或者转换。

（2）使用带有 LOCATION 子句的 CREATE TABLE 语句指向数据文件所在的位置。这样做的前提是数据文件必须是在 HDFS 上，而不是在本地文件系统上。为了防止数据文件会被其他的应用访问、使用或者修改，我们也可以使用 CREATE EXTERNAL TABLE 语法通过创建外部表的方式指向这些数据文件。

（3）如果已经有现成的 Parquet 表了，我们可以直接把数据文件拷贝到表对应的数据目录中，然后使用 REFRESH 语句刷新表对应的元数据信息。在复制 Parquet 数据文件时不要使用-put 或者-cp 选项，而要使用 hdfs distcp -pb 命令进行复制，这个命令可以可以保证在复制过程中使用原始文件的块大小进行复制。

如果外部系统产生的不是 Parquet 格式的文件，而是其他格式的，我们就要综合使用上述技术。首先，我们使用 LOAD DATA 或者 CREATE EXTERNAL TABLE … LOCATION 语句加载到与文件原始格式匹配的 Impala 表中，然后使用 INSERT … SELECT 将表中的数据拷贝到 Parquet 格式的表中。

向 Parquet 表中加载数据是一个消耗内存的操作，因为只要当加载的数据在内存中缓存了 1GB 之后，这些数据被重组也压缩之后才会写到 HDFS 上。而如果是向 Parquet 分区表中写入数据，每个分区键值组合就会生成一个单独的数据文件，同时也就需要多个 1GB 的内存写缓冲区。

当我们向 Parquet 表插入数据时，Impala 会在节点间重新分布数据以减少内存消耗。即使这样在加载数据时我们可能仍然需要临时增加 Impala 可以独占的内存大小。如果无法增加 Impala 独占的内存大小，我们也可以把加载操作拆分成多个 INSERT 语句进行插入。

7.3.3　查询性能

Parquet 表的查询性能依赖于我们需要 SELECT 的列的数目，WHERE 条件的区分度等。如下示例中的查询就是一个高效的查询语句：

```
select avg(income) from census_data where state = 'CA';
```

这个查询只涉及了两个列的数据，而其中表是按 state 列做的分区。所以我们在读取数据时不

会读取其他的 state 的数据，而在 state='CA'的数据中只取其中的 income 列进行计算就可以了，大大减少了需要读取和计算的数据量。

同样的道理，下面的这个查询语句效率就会非常低下：

```
select * from census_data;
```

Impala 需要读取所有的数据文件，为表中每一列的每一个行组进行解压，这个查询没有从面向列式的存储方式中获得任何好处。虽然它没有发挥任何 Parquet 数据文件的优势，但是它可能仍然比某些文件存储格式要快。

当 Parquet 表搜集过统计信息之后，Impala 可以针对 Parquet 表进行优化操作，尤其是可以进行关联查询的优化操作。

Parquet 表分区

Parquet 表非常适合于那些表本身包含很多列，但是仅对很少的一些列进行的查询。当对 Parquet 进行了分区之后，性能会得到进一步的提升。Impala 会忽略 WHERE 条件中没有的分区，仅对指定的分区进行数据文件读取操作。

想 Parquet 分区表中插入数据是一个特别消耗资源的操作，因为每个 Impala 节点都可能为一种分区键值的组合写入一个单独的数据文件。在 HDFS 上同时打开的文件数收参数“transceivers”的限制。为了避免超出这个限制，考虑使用如下技术：

（1）使用不同的 INSERT 语句加载特定的分区，比如指定 PARTITION (year=2010)。

（2）增加参数“transceiver”的值，在有的版本也拼成“xceivers”。为 HDFS 配置文件 hdfs-site.xml 中的 dfs.datanode.max.transfer.threads 配置一个合适的值。比如，如果我们要想一个按照年、月、日分区的表中加载 12 年的数据，使用 4096 都不算大。

（3）在从源表拷贝数据之前对表使用 COMPUTE STATS 搜集统计信息。这样做是希望 Impala 能够通过统计信息生成一个更有效的执行计划。

7.3.4 Snappy/Gzip 压缩

当我们使用 INSERT 向 Parquet 表插入数据时，可以通过参数 PARQUET_COMPRESSION_CODEC 控制压缩的方式。这个参数允许的值有 snappy（该值为默认值），gzip 和 none。这个参数的值可以使用大写或者小写。如果这个参数使用了一个 Impala 无法识别的值，所有的 Impala 查询都会报错。

1. Snappy 压缩示例

默认情况下，Parquet 表使用 Snappy 压缩。它对大批量数据的压缩和解压的速度是相当快的。为了使用 Snappy 压缩，需要在插入数据之前设置 PARQUET_COMPRESSION_CODEC 参数：

```
[hadoop-cs1:21000] > create database parquet_compression;
```

```
[hadoop-cs1:21000] > use parquet_compression;
[hadoop-cs1:21000] > create table parquet_snappy like raw_text_data;
[hadoop-cs1:21000] > set PARQUET_COMPRESSION_CODEC=snappy;
[hadoop-cs1:21000] > insert into parquet_snappy select * from raw_text_data;
Inserted 1000000000 rows in 181.98s
```

2. Gzip 压缩示例

如果我们想拥有更高的压缩比，可以使用 Gzip 压缩，但是它在压缩和解压时也会消耗更多的 CPU 资源。与 Snappy 压缩类似，我们也需要设置 PARQUET_COMPRESSION_CODEC 参数：

```
[hadoop-cs1:21000] > create table parquet_gzip like raw_text_data;
[hadoop-cs1:21000] > set PARQUET_COMPRESSION_CODEC=gzip;
[hadoop-cs1:21000] > insert into parquet_gzip select * from raw_text_data;
Inserted 1000000000 rows in 1418.24s
```

3. 不压缩示例

如果我们的数据量不大，或者想避免压缩和解压时的 CPU 负载，我们可以将 PARQUET_COMPRESSION_CODEC 参数设置为 none，不使用任何压缩方式：

```
[hadoop-cs1:21000] > create table parquet_none like raw_text_data;
[hadoop-cs1:21000] > insert into parquet_none select * from raw_text_data;
Inserted 1000000000 rows in 146.90s
```

4. Parquet 压缩对比测试示例

这个示例中我们使用人造的数据使用 Snappy，Gzip 和 none 三种参数的情况下的数据压缩和查询速度的对比。而我们在测试时最好使用生产环境中的真实数据进行对比测试，数据的压缩比和查询速度在很大程度上依赖于数据本身的特点。

数据压缩情况对比如下：

```
$ hdfs dfs -du -h /user/hive/warehouse/parquet_compression.db
23.1 G /user/hive/warehouse/parquet_compression.db/parquet_snappy
13.5 G /user/hive/warehouse/parquet_compression.db/parquet_gzip
32.8 G /user/hive/warehouse/parquet_compression.db/parquet_none
```

通过上述数据我们可以看到数据从不压缩到 Snappy 压缩，大约节省了三分之一的空间，而从 Snappy 到 Gzip 压缩，大约节省了二分之一的空间。

由于 Parquet 数据文件通常都是 1GB 大小，每个目录中的数据文件的个数也会对空间占用有一定影响。

压缩比和效率就像是天平的两端，压缩比越高，压缩和解压的时间越长，不做压缩占用的空间最大，但是不会消耗压缩解压的时间。

对某列做聚集查询的时间对比如下：

```
[hadoop-cs1:21000] > desc parquet_snappy;
Query finished, fetching results ...
+-----------+---------+---------+
| name | type | comment |
+-----------+---------+---------+
| id | int | |
| val | int | |
| zfill | string | |
| name | string | |
| assertion | boolean | |
+-----------+---------+---------+
Returned 5 row(s) in 0.14s
[hadoop-cs1:21000] > select avg(val) from parquet_snappy;
Query finished, fetching results ...
+-----------------+
| _c0 |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 4.29s
[hadoop-cs1:21000] > select avg(val) from parquet_gzip;
Query finished, fetching results ...
+-----------------+
| _c0 |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 6.97s
[hadoop-cs1:21000] > select avg(val) from parquet_none;
Query finished, fetching results ...
+-----------------+
| _c0 |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 3.67s
```

可以看到不压缩运行速度最快，而使用压缩时 Snappy 比 Gzip 压缩更快。

5. 拷贝 Parquet 数据文件示例

该示例演示如何对使用了不同压缩方式的文件进行读取操作。示例中我们将前面示例中的 PARQUET_SNAPPY、PARQUET_GZIP 和 PARQUET_NONE 三张表的数据文件拷贝到一个新的 Impala 表 PARQUET_EVERYTHING 对应的数据目录中。这三张表每张表有 10 亿条记录，我们最终的新表可以对拷贝过来的不同压缩方式的文件进行读取，最终查询出 30 亿条记录。

首先，我们创建一张空表：

```
[hadoop-cs1:21000] > create table parquet_everything like parquet_snappy;
Query: create table parquet_everything like parquet_snappy
```

表创建完毕之后，Impala 就在 HDFS 上建立了相应的数据目录。我们将之前三张表的数据文件拷贝到该数据目录中：

```
$ hdfs distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_snappy \
/user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hdfs distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_gzip \
/user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hdfs distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_none \
/user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
```

在 impala-shell 中，我们使用 REFRESH 操作刷新新表的元数据信息，然后就可以对这张表进行读取操作了：

```
[hadoop-cs1:21000] > refresh parquet_everything;
Query finished, fetching results ...
Returned 0 row(s) in 0.32s
[hadoop-cs1:21000] > select count(*) from parquet_everything;
Query finished, fetching results ...
+------------+
| _c0 |
+------------+
| 3000000000 |
+------------+
Returned 1 row(s) in 8.18s
[hadoop-cs1:21000] > select avg(val) from parquet_everything;
Query finished, fetching results ...
+-----------------+
```

```
| _c0 |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 13.35s
```

7.3.5 与其他组件交换 Parquet 数据文件

自 CDH 4.5 起，我们就可以通过 Hive，Pig 或者 MapReduce 读写 Parquet 数据文件了。在早期版本中，我们不能通过 Impala 创建 Parquet 数据文件，只能通过 Hive 来创建。在 Impala 1.1.1 或者更高版本中，我们可以通过如下命令修改表对应的数据文件格式：

```
ALTER TABLE table_name SET FILEFORMAT PARQUET;
```

如果我们使用的 Impala 版本低于 1.1.1，我们需要在 Hive 中使用如下命令来设置表对应的数据文件格式：

```
ALTER TABLE table_name SET SERDE 'parquet.hive.serde.ParquetHiveSerDe';
ALTER TABLE table_name SET FILEFORMAT
INPUTFORMAT "parquet.hive.DeprecatedParquetInputFormat"
OUTPUTFORMAT "parquet.hive.DeprecatedParquetOutputFormat";
```

对于 Parquet 表，Impala 支持标量数据类型，但是不支持向 Map、Array 这样的组合类型和内嵌类型。如果 Parquet 表的列使用了不支持的数据类型，Impala 就不能正确的访问这张表。

如果需要拷贝 Parquet 数据文件，一定要使用 hadoop distcp -pb 命令保证拷贝过程中保持原始数据块的大小。另外可以使用命令 hdfs fsck -blocks HDFS_path_of_impala_table_dir 检查目标文件是否保持了原始文件的块大小。

7.3.6 Parquet 数据文件组织方式

虽然 Parquet 数据文件是面向列存储的，但是也不是一个列一个数据文件。Parquet 将一行所有字段都存在同一个数据文件中。一个 Parquet 数据文件的最大大小是 1GB，我们要确保 IO 和网络传输的参数与这个大小想匹配。

在这 1GB 的数据文件内，所有的行被重新组织，所有行的第一列的值相邻存放，然后是第二列，第三列，以此类推。把同一列的值集中存放，因为数据类型都是一样的，所以也使针对单列的压缩更有效率。

当 Impala 需要读取某列数据时，只需要打开数据文件只读取被连续存放在一起的该列相关的数据就可以了。如果查询中或者 WHERE 条件中还出现了其他列，这些不同的列也可以在同一个数据文件中找到。

如果 INSERT 插入的数据没有 1GB，那么最终的数据文件可能会小一些。如果我们要使用多个 ETL 作业来进行 INSERT 操作，需要尽量保证数据量是 1GB 左右，或者是 1GB 的倍数。

1. RLE 和数据字典编码

基于生产环境的真实数据值，Parquet 自动采用向游程编码（RLE）或者数据字典编码方式对数据进行自动压缩。在数据以上述的技术对值进行相对紧凑的编码之后，我们还可以进一步使用压缩算法对其进行压缩。Parquet 支持三种方式：Snappy、Gzip 和无压缩。虽然 Parquet 文件的规范也支持 LZO 压缩，但是目前 Impala 的 Parquet 文件还不支持该压缩算法。

Impala 对 Parquet 文件中的数据值使用游程编码（RLE）或者数据字典编码进行压缩编码，对整个 Parquet 数据文件本身使用 Snappy 或者 Gzip 进行压缩。无论是对数据值进行那种编码，还是对数据文件进行压缩都是 Impala 自动进行的，这大大节省了时间。比如，数据字典编码会对重复出现的长字符串，后续的长字符串仅会存储指向第一个字符串的指针而非存储字符串本身，以节省存储空间。使用数据字典编码时，一定要保证编码的列的非重复值不得超过 16384。而游程编码（RLE）着重处理连续的重复值序列，如果一个值连续多行出现，那么使用游程编码（RLE）可以将其缩写为值和连续出现的次数。

2. 压缩数据文件

如果我们重用已有的表结构或者针对 Parquet 表的 ETL 过程，我们可能会遇到有很多小的数据文件的情况，这将大大降低查询的效率。以下两种情况可能产生此种情况：

（1）源数据数据量过小

在 N 个节点的集群中，INSERT 语句在每个节点都会产生一个数据文件，比如我们源表的数据量有 1GB，我们有 10 个数据节点，那么平均到每个节点上的数据量就只有 100M，这些数据再经过 Parquet 的压缩，就变成了我们所说的小文件。

```
insert into parquet_table select * from text_table;
```

（2）单个分区包含的数据量过小

虽然源数据的数据量非常庞大，但是经过分区之后的单个分区包含的数据量过小。

```
insert into partitioned_parquet_table partition (year, month, day)
select year, month, day, url, referer, user_agent, http_code, response_time
from web_stats;
```

我们有以下技术可以在 INSERT 操作时产生大文件，或者合并已存在的小文件：

（1）当我们向 Parquet 分区表插入数据时，使用将分区键值指定为常量的静态分区，为每个分区使用一个单独的 INSERT 语句。

（2）我们可以在 INSERT 或者 CREATE TABLE AS SELECT 语句时，将参数 NUM_NODES 指定为 1。默认情况下，CREATE TABLE AS SELECT 或者 INSERT 语句在每个节点可以产生一个或者多个数据文件。如果要写的数据的数据量不大，那么势必会产生很多小文件。将参数

NUM_NODES 设置为 1 可以关闭写操作的分布式特性，这样写操作相当于在单个节点上执行，只会产生少量的大文件。

（3）减少分区列的数目，这样就相当于增大了单个分区的数据量。

（4）不要指望 Impala 在写时会把数据填满一个 Parquet 数据块。在插入时，Impala 会保守估计每个 Parquet 文件将要写入的数据量。另外，通常在内存中 1GB 的未压缩数据，经过编码和压缩之后到磁盘上可能要小得多，可能要就几百兆。最终数据的多少要依赖于数据本身的特点。因此，如果 1GB 的文件被拆分写到了两个 Parquet 文件中，是正常现象。

（5）如果我们已经产生了很多小的数据文件的情况，我们可以根据前面几点提到的技术通过 CREATE TABLE AS SELECT 或者 INSERT 等将数据插入一张新的表。

如何避免因为要修改表的名称而需要将数据重新拷贝写到一张新的表？那就是使用视图技术。通过改变视图定义可以让我们方便的改变视图指向的基表。

最开始时我们创建的视图基于有很多小文件的表：

```
create view production_table as select * from table_with_many_small_files;
```

我们经过上述的技术将小文件合并之后产生了新的表。我们将视图指向新的表：

```
alter view production_table as select * from table_with_few_big_files;
```

那么原有的 SQL 语句不需要任何变化就可以进行查询了：

```
select * from production_table where c1 = 100 and c2 < 50 and ...;
```

3. 模式进化

这里的模式进化指的是使用 ALTER TABLE … REPLACE COLUMNS 来改变 Parquet 表的数据类型、列数等。我们可以执行的变更包括如下几种：

（1）Impala 中的 ALTER TABLE 语句从来不会改变表对应的数据文件本身。从 Impala 的视角来看，模式进化就是使用新的表的对应来解释同一个数据文件而已。比如：某些列的数据类型的变化可能更符合应用的真实意义。而如果在新的表的定义中使用了数据文件中无法正常转换的类型，查询将会报错。

（2）INSERT 语句总是依据最新的表定义进行插入操作。如果我们在 INSERT 的过程中改变列的定义可能会导致底层数据的存储依据了不同的表定义。

（3）如果我们通过 ALTER TABLE … REPLACE COLUMNS 为表定义了数据文件中没有的列，那么在查询中，新的列的值都为 NULL。

（4）如果我们通过 ALTER TABLE … REPLACE COLUMNS 为表定义的列没有数据文件中的列多，那么在查询中，将会自动忽略数据文件中多余的列的值。

（5）Parquet 使用 32 位的整型值来存储 TINYINT、SMALLINT、INT 类型的值。

这意味着我们很容易进行 TINYINT、SMALLINT、INT 各类型之间进行转换，因为在底层数据的存储中是相同的。

如果我们试图将一个范围大的类型强制转换为一个范围小的类型，仍然可能产生溢出错误。

我们不能将 TINYINT、SMALLINT、INT 转换为 BIGINT。即使执行 ALTER TABLE 时可以成功，但是在真正查询时仍然会报错。

同样的，任何其他类型的转换都可能引起查询时的转换错误。比如：将 INT 转换为 STRING，FLOAT 转换为 DOUBLE，TIMESTAMP 转换为 STRING，DECIMAL(9，0)转换为 DECIMAL(5，2)等。

7.4 Avro

Impala 支持表使用 Avro 文件格式的数据文件。在 Impala 1.4.0 或者更高的版本，可以直接创建 Avro 表，但是如果使用的是之前的版本，我们必须通过 Hive 来完成创建表和插入数据的操作。Avro 文件类型的特点如下表所示：

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
Avro	结构化	Snappy，GZIP，deflate，BZIP2	是。在 Impala 1.4.0 或者更高版本可以，但是在低版本中必须通过 Hive 创建	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载

7.4.1 创建 Avro 表

为了创建 Avro 表，我们需要在 Impala 或者 Hive 中执行带有 STORED AS AVRO 子句的语句。如果使用 Impala，我们必须在语句中指定列定义信息。如果使用 Hive，我们可以忽略列定义信息。

如下示例我们使用了两种方式来定义 Avro 表 一种是将 JSON 内嵌到语句中，一种是将 JSON 存放在 HDFS 上。

```
[hadoop-cs1:21000] > CREATE TABLE impala_avro_table
> (name BOOLEAN, int_col INT, long_col BIGINT, float_col FLOAT,
double_col DOUBLE, string_col STRING, nullable_int INT)
> STORED AS AVRO
> TBLPROPERTIES ('avro.schema.literal'='{
> "name": "my_record",
> "type": "record",
> "fields": [
> {"name":"bool_col", "type":"boolean"},
> {"name":"int_col", "type":"int"},
> {"name":"long_col", "type":"long"},
```

```
> {"name":"float_col", "type":"float"},
> {"name":"double_col", "type":"double"},
> {"name":"string_col", "type":"string"},
> {"name": "nullable_int", "type": ["null", "int"]}]}');
[hadoop-cs1:21000] > CREATE TABLE avro_examples_of_all_types (
> id INT,
> bool_col BOOLEAN,
> tinyint_col TINYINT,
> smallint_col SMALLINT,
> int_col INT,
> bigint_col BIGINT,
> float_col FLOAT,
> double_col DOUBLE,
> date_string_col STRING,
> string_col STRING,
> timestamp_col TIMESTAMP
> )
> STORED AS AVRO
> TBLPROPERTIES
('avro.schema.url'='hdfs://localhost:8020/avro_schemas/alltypes.json');
```

如下示例是在 Hive 中创建 Avro 表的示例：

```
hive> CREATE TABLE hive_avro_table
> ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
>                    STORED                    AS                    INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
> OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
> TBLPROPERTIES ('avro.schema.literal'='{
> "name": "my_record",
> "type": "record",
> "fields": [
> {"name":"bool_col", "type":"boolean"},
> {"name":"int_col", "type":"int"},
> {"name":"long_col", "type":"long"},
> {"name":"float_col", "type":"float"},
> {"name":"double_col", "type":"double"},
> {"name":"string_col", "type":"string"},
> {"name": "nullable_int", "type": ["null", "int"]}]}');
```

其中每个字段对应表的一个列。

大多数的列类型可以通过 Avro 向 Impala 直接映射，但是也有特殊情况：

（1）在 Avro 中 DECIMAL 是一个有 logicalType 属性的 BYTE 类型，我们可以为其指定精度和位数。只有在 CDH5 的 Avro 表中可以使用 DECIMAL 类型。

（2）在 Avro 中的 long 类型与 Impala 中的 BIGINT 相对应。

如果我们在 Hive 中创建了一张 Avro 表，则必须在 impala-shell 中使用 INVALIDATE METADATA table_name 来更新 Impala 的元数据信息。当我们连接到任何一个 Impala 节点时，catalog 服务会将元数据信息广播到其他 Impala 节点。另外，如果通过 Hive 进行了 LOAD DATA 或者 INSERT 等操作，或者手动将数据文件拷贝到了 Avro 表的数据目录，我们都需要通过 REFRESH table_name 语句刷新元数据信息。

如果我们有现成的 Avro 文件，我们可以在 Impala 或者 Hive 中使用 LOAD DATA 加载数据。

7.4.2　使用 Hive 创建的 Avro 表

如果我们想在 Impala 中使用通过 Hive 创建的 Avro 表，那么表的数据类型必须与 Impala 兼容。这些数据类型**不包括**：

（1）复杂类型：array、map、record、struct，除[supported_type,null]或者[null,supported_type]之外的 union 类型。

（2）像 enum、bytes、fixed 等 Avro 特有的类型。

（3）任何在数据类型部分介绍过之外的标量数据类型。

Impala 只支持 BOOLEAN、INT、LONG、FLOAT、DOUBLE、STRING 类型，或者这些类型和 NULL 的 UNION 类型，如[“string”,”null”]。

7.4.3　通过 JSON 指定 Avro 模式

我们可以将 JSON 形式的列定义内嵌到 CREATE TABLE 语句中。但是如果指定的长度如果超过了 Hive 元数据中列宽度的限制，将无法创建成功。如果要解决这个问题，我们可以将 JSON 文件存放到 HDFS 上，并在定义表属性时指向该文件：

```
tblproperties ('avro.schema.url'='hdfs//your-name-node:port/path/to/schema.json');
```

7.4.4　启用压缩

如果需要对 Avro 数据文件进行压缩，需要在 Hive shell 中指定相关参数：

```
hive> set hive.exec.compress.output=true;
```

```
hive> set avro.output.codec=snappy;
```

7.4.5 模式进化

自 Impala 1.1 起，我们可以对 Avro 数据文件进行模式进化。针对同一张表的数据文件，可以使用略微不同的数据定义。这样做的前提是老的数据类型和新的数据类型必须兼容，比如我们可以把定义为 INT 的列修改为 BIGINT 或者 FLOAT。

当 Avro 数据文件或者数据列没有被查询使用到时，Impala 不会检查其一致性。比如，执行查询 SELECT c1，c2 FROM t1，如果 t3 列存在不兼容的情况，该语句不会报错。针对分区表也是类似的，Impala 不会检查没有使用到的分区的数据文件不兼容的情况。

在 Hive DDL 语句中，我们可以指定表的 avro.schema.literal 属性（当表的定义很短时使用）和 avro.schema.url 属性（当表的定义很长，将 JSON 文件放到 HDFS 上，通过 URL 引用该文件的位置）。

如下示例使用 Avro 格式创建表，并插入数据：

```
CREATE TABLE avro_table (a string, b string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
TBLPROPERTIES (
'avro.schema.literal'='{
"type": "record",
"name": "my_record",
"fields": [
{"name": "a", "type": "int"},
{"name": "b", "type": "string"}
]}');
INSERT OVERWRITE TABLE avro_table SELECT 1, "avro" FROM functional.alltypes LIMIT
1;
```

Avro 表有了数据之后，我们就可以通过 impala-shell 进行查询：

```
-- [hadoop-cs1:21000] > select * from avro_table;
-- Query: select * from avro_table
-- Query finished, fetching results ...
-- +---+------+
-- | a | b |
-- +---+------+
```

```
-- | 1 | avro |
-- +---+------+
```

在 Hive shell 中更改列数据类型并添加一个有默认值的新列：

```
-- Promote column "a" from INT to FLOAT (no need to update Avro schema)
ALTER TABLE avro_table CHANGE A A FLOAT;
-- Add column "c" with default
ALTER TABLE avro_table ADD COLUMNS (c int);
ALTER TABLE avro_table SET TBLPROPERTIES (
'avro.schema.literal'='{
"type": "record",
"name": "my_record",
"fields": [
{"name": "a", "type": "int"},
{"name": "b", "type": "string"},
{"name": "c", "type": "int", "default": 10}
]}');
```

在 impala-shell 中我们可以查询表最新的定义。因为表定义是在 Impala 之外进行更改的，所以需要更新表的元数据信息，让 Impala 能够识别到最新的表定义变更：

```
-- [hadoop-cs1:21000] > refresh avro_table;
-- Query: refresh avro_table
-- Query finished, fetching results ...
-- Returned 0 row(s) in 0.23s
-- [hadoop-cs1:21000] > select * from avro_table;
-- Query: select * from avro_table
-- Query finished, fetching results ...
-- +---+------+----+
-- | a | b | c |
-- +---+------+----+
-- | 1 | avro | 10 |
-- +---+------+----+
-- Returned 1 row(s) in 0.14s
```

7.5 RCFile

Impala 支持使用 RCFile 格式的数据文件，其特点如下表所示。

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
RCFile	结构化	Snappy，GZIP，deflate，BZIP2	是	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载

7.5.1 创建 RCFile 表和加载数据

如果我们已经在 Impala 之外生成了 RCFile 数据文件，那么首先我们应该创建一张支持 RCFile 格式的表，在 impala-shell 中使用命令：

```
create table rcfile_table (column_specs) stored as rcfile;
```

因为创建表之后，目前 Impala 不能向其中写入数据。所以加载数据的过程需要通过 Hive 或者 Impala 之外的机制完成，然后在通过 REFRESH table_name 语句刷新 Impala 的元数据信息，再通过 Impala 对其进行查询。

如下示例演示了如何在 Impala 中创建 RCFile 表，通过 Hive 加载数据，并通过 Impala 进行查询的过程：

```
$ impala-shell -i localhost
[hadoop-cs1:21000] > create table rcfile_table (x int) stored as rcfile;
[hadoop-cs1:21000] > create table rcfile_clone like some_other_table stored as
rcfile;
[hadoop-cs1:21000] > quit;
$ hive
hive> insert into table rcfile_table select x from some_other_table;
3 Rows loaded to rcfile_table
Time taken: 19.015 seconds
hive> quit;
$ impala-shell -i localhost
[hadoop-cs1:21000] > select * from rcfile_table;
Returned 0 row(s) in 0.23s
[hadoop-cs1:21000] > -- Make Impala recognize the data loaded through Hive;
[hadoop-cs1:21000] > refresh rcfile_table;
[hadoop-cs1:21000] > select * from rcfile_table;
```

```
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.23s
```

7.5.2　启用压缩

我们可以对一个 RCFile 表进行压缩。启用压缩在大多数情况下可以带来性能提升。如下示例演示了如何启用 Snappy 压缩：

```
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> INSERT OVERWRITE TABLE new_table SELECT * FROM old_table;
```

如果要对分区表使用压缩，我们还需要设置额外的参数：

```
hive> CREATE TABLE new_table (your_cols) PARTITIONED BY (partition_cols) STORED
AS
new_format;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE new_table PARTITION(comma_separated_partition_cols)
SELECT
* FROM old_table;
```

将两个过程综合考虑，对普通表的设置过程如下：

```
hive> CREATE TABLE tbl_rc (int_col INT, string_col STRING) STORED AS RCFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
```

```
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_rc SELECT * FROM tbl;
```

将两个过程综合考虑，对分区表的设置过程如下：

```
hive> CREATE TABLE tbl_rc (int_col INT, string_col STRING) PARTITIONED BY (year
INT)
STORED AS RCFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_rc PARTITION(year) SELECT * FROM tbl;
```

7.6 SequenceFile

Impala 支持使用 SequenceFile 格式的数据文件，其特点如下表所示。

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
SequenceFile	结构化	Snappy，GZIP，deflate，BZIP2	是	否。通过 LOAD DATA 加载已经具有正确格式的文件，或者通过 Hive 使用 INSERT 的方式加载

7.6.1 创建和加载数据

在 impala-shell 中创建 SequenceFile 表，创建示例如下：

```
create table sequencefile_table (column_specs) stored as sequencefile;
```

虽然可以在 Impala 中创建 SequenceFile 数据文件格式的表，但是却无法在 Impala 中插入数据。我们还需要通过 Hive shell 向该表中加载数据，然后再使用 REFRESH table_name 刷新 Impala 元数据信息，在通过 Impala 对其进行查询。

如下示例演示了如何在 Impala 中创建表，在 Hive 中插入数据，并通过 Impala 进行查询的过程：

```
$ impala-shell -i localhost
[hadoop-cs1:21000] > create table seqfile_table (x int) stored as sequencefile;
[hadoop-cs1:21000] > create table seqfile_clone like some_other_table stored as
sequencefile;
[hadoop-cs1:21000] > quit;
$ hive
hive> insert into table seqfile_table select x from some_other_table;
3 Rows loaded to seqfile_table
Time taken: 19.047 seconds
hive> quit;
$ impala-shell -i localhost
[hadoop-cs1:21000] > select * from seqfile_table;
Returned 0 row(s) in 0.23s
[hadoop-cs1:21000] > -- Make Impala recognize the data loaded through Hive;
[hadoop-cs1:21000] > refresh seqfile_table;
[hadoop-cs1:21000] > select * from seqfile_table;
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.23s
```

7.6.2　启用压缩

我们可以对 SequenceFile 表启用压缩。在大多数情况下，启用压缩会带来性能提升。对 SequenceFile 启用压缩的过程与 RCFile 表类似，对于普通表压缩的过程：

```
hive> create table TBL_SEQ (int_col int, string_col string) STORED AS SEQUENCEFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_seq SELECT * FROM tbl;
```

对于分区表，压缩的过程：

```
hive> CREATE TABLE tbl_seq (int_col INT, string_col STRING) PARTITIONED BY (year
INT)
STORED AS SEQUENCEFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_seq PARTITION(year) SELECT * FROM tbl;
```

7.7 HBase

Impala 支持以通用的 SQL 语句的方式对 HBase 表进行查询，HFile 文件类型的特点如下：

文件类型	格式	压缩编码	Impala 是否可直接创建	Impala 是否可直接插入
HFile（HBase）	结构化	GZIP,LZO,Snappy	否。必须通过 Hive 创建	是。支持 INSERT 操作

默认的 Impala 表使用存储在 HDFS 上的数据文件，这样的表对于批量数据加载，对表进行全表扫描的查询非常有优势。而 HBase 表对基于键值的单行查询或者范围查询支持得更好，非常适合用于 OLTP 系统。Impala 表可以将 HBase 表作为其底层存储使用，大大扩展了其使用的范围。

从 Impala 的视角来看，HBase 是一个值包含了很多字段的一个键值对存储系统。其中的键映射到 Impala 表的一个列，而其中的值映射到 Impala 表的其他列。

当我们使用基于 HBase 的 Impala 表时：

（1）在 Impala 侧创建表时可以通过 Hive shell 来创建，因为目前 Impala 的 CREATE TABLE 语句的很多语法并不支持。

- 我们需要使用 Hive 的创建表的 CREATE TABLE 语句的 STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'子句来创建基于 HBase 的表。
- 我们可以使用 Hive 的创建表 CREATE TABLE 语句的 TBLPROPERTIES("hbase.table.name" = "table_name_in_hbase")来指向一个已经存在的 HBase 表。

（2）我们可以使用#string 关键字将 HBase 行健作为字符串来定义，或者映射到一个 STRING 列。

（3）因为 Impala 和 Hive 共享同一个元数据库，所以只要我们通过 Hive 创建了表，就可以通过 Impala 来进行查询或者插入操作（在 Hive 中创建了表后，需要在 impala-shell 中执行 INVALIDATE METADATA 语句让 Impala 可以识别到新的表）。

（4）在对表进行查询时最好指定与 HBase 行键对应的列的过滤条件，而避免使用全表扫描。对普通 Impala 使用全表扫描是可以的，但是对基于 HBase 的 Impala 表进行全表扫描是一个非常低效的操作。

7.7.1　支持的 HBase 列类型

HBase 以一种称为“bit bucket”的方式工作，在这个意义上说，HBase 不会对键或者值做任何转换和处理，所有的关于数据类型的处理都是在 Impala 侧进行的。

从性能上考虑，对基于 HBase 的 Impala 表的查询需要指定和 HBase 行键相关的 WHERE 条件。当我们通过 Hive 创建表时使用 STRING 类型映射了 HBase 的行键，那么 Impala 就能通过谓词推进的方式将针对映射列的=、<、BETWEEN、IN 查询翻译成针对 HBase 表行键的条件查询，而我们知道在 HBase 中针对行键的查询效率是极高的。需要注意的是，这种谓词推进的优化只有在映射列被定义为 STRING 时才有效。

自 Impala 1.1 起，Impala 也支持读写在 Hive 使用#binary 关键字创建的数据类型。在 HBase 中，我们也经常使用二进制类型代替数字类型来减少存储的数据量。即使这样，我们仍然强烈建议将 HBase 的行键定义为 STRING 类型，以便能够在 Impala 中对 HBase 表进行快速查询。

7.7.2　性能问题

Impala 通过 JNI(Java Native Interface)使用 HBase 客户端 API 查询存储在 HBase 中的数据。Impala 查询不会直接读取 HBase 数据文件。Impala 查询基于 HBase 的表需要额外的通信负载。对于一张 Impala 表，我们选择使用 HDFS 作为存储还是使用 HBase 作为存储，是需要慎重考虑的。对于基于 HBase 的 Impala 表，为了构造一个基于 HBase 的高效的查询 SQL，我们需要：

（1）使用 HBase 表的查询需要返回单行或者某个范围的数据行，而不是返回整张表。如果一个查询没有 WHERE 条件对数据进行过滤，那么基本可以判定这个语句的执行效率会很低。

（2）在数据仓库中，我们应用的典型的场景是一张非常大的事实表和几张比较小的维度表进行关联操作。在这个场景中，对事实表的查询基本是对全表或者某个分区的全扫描，而对维度表的查询则是根据查询条件过滤出需要维度的单行或者范围查询。在这种场景中，我们建议事实表使用 HDFS 存储，而维度表使用基于 HBase 的表存储。

应用到 HBase 行键的查询谓词可以很好的限定查询的范围。但是如果行键没有映射到 STRING 列，在底层行键就无法正确的排序，查询时的比较操作符将无法正确使用。

当谓词使用了非行键的列时，查询相当于向 HBase 发送了一个 SingleColumnValueFilters，跟使用普通的 Impala 表相比，查询会有一定的性能提升。因为如果使用非行键作为谓词，即使我们最终返回的数据只有很少的行，但是它也需要扫描 HBase 的整张表。

1. 执行计划

我们可以通过 Impala 的执行计划来查看一个查询执行的细节，并且通过执行计划提供的信息来初步判定一个查询的效率。

```
describe hbase_table;
Query: describe hbase_table
+-----------------------+--------+---------+
| name | type | comment |
+-----------------------+--------+---------+
| cust_id | string | |
| birth_year | string | |
| never_logged_on | string | |
| private_email_address | string | |
| year_registered | int | |
+-----------------------+--------+---------+
```

该表第一列 cust_id 为 HBase 表的行键。根据之前的描述，我们需要把这列定义为 STRING 列。而且数据列 BIRTH_YEAR、NEVER_LOGGED_ON 原有的类型分别为 INT 和 BOOLEAN，现在也被定义为 STRING 列，因为 Impala 对 STRING 类型的处理比 HBase 更有效率。为了比较这种效率上的差别，我们故意把列 YEAR_REGISTERED 保持为原有的 INT 类型。

使用 HBase 的典型的场景就是对行键进行一个等值比较的查询：

```
explain select count(*) from hbase_table where cust_id = 'some_user@example.com';
+----------------------------------------------------------------------------
-------+
| Explain String
|
+----------------------------------------------------------------------------
-------+
| Estimated Per-Host Requirements: Memory=1.01GB VCores=1
|
| WARNING: The following tables are missing relevant table and/or column statistics.
|
| hbase.hbase_table
|
|
```

```
|
| 03:AGGREGATE [MERGE FINALIZE]
|
| | output: sum(count(*))
|
| |
|
| 02:EXCHANGE [PARTITION=UNPARTITIONED]
|
| |
|
| 01:AGGREGATE
|
| | output: count(*)
|
| |
|
| 00:SCAN HBASE [hbase.hbase_table]
|
| start key: some_user@example.com
|
| stop key: some_user@example.com\0
|
+-----------------------------------------------------------------------------
-------+
```

另外一个典型的场景是对行键进行范围查询。如下示例中除了对行键进行范围查询，还对一个非行键进行了等值查询。Impala 可以将对这个非行键的等值查询推送到 HBase 中去进行比较，在执行计划的 hbase filters 列中我们可以看到这个信息。在 HBase 中进行这种过滤远远比将数据传送到 Impala 中再进行过滤来得高效。

```
explain select count(*) from hbase_table where cust_id between 'a' and 'b'
and never_logged_on = 'true';
+-----------------------------------------------------------------------------
-------+
| Explain String
|
+-----------------------------------------------------------------------------
-------+
```

```
...
| 01:AGGREGATE
|
| | output: count(*)
|
| |
|
| 00:SCAN HBASE [hbase.hbase_table]
|
| start key: a
|
| stop key: b\0
|
| hbase filters: cols:never_logged_on EQUAL 'true'
|
+------------------------------------------------------------------------------
-------+
```

如下的查询在 WHERE 条件中只有针对一个非行键列的等值比较，Impala 必须要扫描整张 HBase 表才能返回结果。Impala 只有将 HBase 中的列声明为 STRING 时，才会将 SQL 中的谓词推进到 HBase 中处理。这个查询中的 YEAR_REGISTERED 保留了原来的 INT 类型。在执行计划的 predicates 行表示针对数据的过滤是在数据全部传到 Impala 中处理过之后进行的。这样的查询效率会非常低。

```
explain select count(*) from hbase_table where year_registered = 2010;
+------------------------------------------------------------------------------
-------+
| Explain String
|
+------------------------------------------------------------------------------
-------+
...
| 01:AGGREGATE
|
| | output: count(*)
|
| |
|
| 00:SCAN HBASE [hbase.hbase_table]
```

```
  |
  | predicates: year_registered = 2010
  |
  +------------------------------------------------------------------
-------+
```

如果行键列跟一个非常量值进行比较，那么即使行键列在 Impala 中映射到了 STRING 列，即使也使用了等值操作符，Impala 也必须要扫描整张 HBase 表来进行处理。

```
  explain select count(*) from hbase_table where cust_id = private_email_address;
  +------------------------------------------------------------------
-------+
  | Explain String
  |
  +------------------------------------------------------------------
-------+
  ...
  | 01:AGGREGATE
  |
  | | output: count(*)
  |
  | |
  |
  | 00:SCAN HBASE [hbase.hbase_table]
  |
  | predicates: cust_id = private_email_address |
  +------------------------------------------------------------------
-------+
```

目前，Impala 对于 OR 或者 IN 子句并没有进行特别的优化。如果我们在查询中使用了这两个关键字，Impala 可能会将所有 HBase 表的数据传输到 Impala 中处理完后再进行过滤。

```
  explain select count(*) from hbase_table where
  cust_id = 'some_user@example.com' or cust_id = 'other_user@example.com';
  +------------------------------------------------------------------
-----------+
  | Explain String
  |
  +------------------------------------------------------------------
-----------+
```

```
...
| 01:AGGREGATE
|
| | output: count(*)
|
| |
|
| 00:SCAN HBASE [hbase.hbase_table]
|
|    predicates:    cust_id    =    'some_user@example.com'    OR    cust_id    =
'other_user@example.com'
|
+--------------------------------------------------------------------------------
-----------+
explain select count(*) from hbase_table where
cust_id in ('some_user@example.com', 'other_user@example.com');
+--------------------------------------------------------------------------------
-------+
| Explain String
|
+--------------------------------------------------------------------------------
-------+
...
| 01:AGGREGATE
|
| | output: count(*)
|
| |
|
| 00:SCAN HBASE [hbase.hbase_table]
|
| predicates: cust_id IN ('some_user@example.com', 'other_user@example.com')
|
+--------------------------------------------------------------------------------
-------+
```

那么如何解决这个问题呢？我们可以将上述语句改写为 UNION ALL 连接的没有 OR 和 IN 的查询。

```
explain
```

```
select count(*) from hbase_table where cust_id = 'some_user@example.com'
union all
select count(*) from hbase_table where cust_id = 'other_user@example.com';
+------------------------------------------------------------------------------
-------+
| Explain String
|
+------------------------------------------------------------------------------
-------+
...
| | 04:AGGREGATE
|
| | | output: count(*)
|
| | |
|
| | 03:SCAN HBASE [hbase.hbase_table]
|
| | start key: other_user@example.com
|
| | stop key: other_user@example.com\0
|
| |
|
| 10:MERGE
|
...
| 02:AGGREGATE
|
| | output: count(*)
|
| |
|
| 01:SCAN HBASE [hbase.hbase_table]
|
| start key: some_user@example.com
|
| stop key: some_user@example.com\0
|
```

```
+-----------------------------------------------------------------------
-------+
```

2. 配置 HBase Java 应用参数

如果在我们的 HBase Java 应用中调用了 org.apache.hadoop.hbase.client.Scan 类的 setCacheBlocks 或者 setCaching 方法，为了控制 HBase region 服务器的内存消耗，我们需要 Impala 查询参数来进行同样的设置。比如，当我们对 HBase 表进行一个全表扫描的查询时，我们可以通过关闭参数 HBASE_CACHE_BLOCKS 并设定一个很大的 HBASE_CACHING 来减少内存的使用，加快查询速度。

在 Impala 中配置的等同参数需要在 impala-shell 中进行配置：

```
-- 等同 setCacheBlocks(true) 或者 setCacheBlocks(false).
set hbase_cache_blocks=true;
set hbase_cache_blocks=false;
-- 等同 setCaching(1000).
set hbase_caching=1000;
```

另外，也可以更改 impalad 进程的默认文件 /etc/default/impala，把参数 HBASE_CACHE_BLOCKS 和 HBASE_CACHING 配置在-default_query_options 部分。

7.7.3 适用场景

如下的场景最适合使用 Impala 来查询基于 HBase 的表：

（1）非常大的事实表使用 Impala 存储，非常小的维度表使用基于 HBase 的表存储。事实表使用 Parquet 数据格式存储，适合于对单列或者少数列的指标进行全表扫描，维度表使用 HBase 存储，适合于对维度进行精确定位。

（2）使用 HBase 存储快速增加的数据。HBase 可以有效地处理变化的数据：通过追加的方式向磁盘写入新增数据，对于变更的数据也是以追加的方式新增数据，然后通过时间戳识别返回最新的版本的数据。HBase 对于我们指定的汇总查询有一定优势，而 Impala 可以执行的查询更有普遍性。

（3）存储非常宽的表用于精确查询。这里的宽表指的甚至可能包含几千列。而这种类型的表往往是稀疏的，单列中可能包含很多的 NULL、0、”、空格等。如果对这样的表查询仅获取很少量的记录，可能使用 HBase 更有优势。

另外，基于 HBase 的 Impala 表可以与其他的 Impala 表进行关联操作。

7.7.4 数据加载

Impala 中的 INSERT 语句可以对基于 HBase 的 Impala 表进行数据插入操作。我们知道 INSERT… VALUES 语句对一个普通的 Impala 表进行数据插入是一个效率非常低的操作，同时还会产生很多小文件。但是对于基于 HBase 的 Impala 表，每次的插入操作都是向 HBase 中插入一条记录，这对 HBase 来说是一个高效的操作方式。

当我们使用 INSERT…SELECT 向基于 HBase 的 Impala 表中插入数据时，可能我们最终能够查询到的数据比真正插入的数据要少。这是因为，针对拥有同一个行键的记录，最后插入的记录拥有最大的版本号，查询时 HBase 对同一个行键仅返回具有最大版本号的记录。虽然 HBase 不支持更新操作，但是这种情况相当于变相的实现了 UPDATE 操作。

7.7.5 启用压缩

由于 Impala 对 HBase 的支持是通过引用 HBase 表实现的，所以对基于 HBase 的 Impala 表的压缩，其实就是对 HBase 表本身的压缩。

GZIP 压缩

对 GZIP 的压缩是通过在创建 HBase 表时指定 COMPRESSION => 'GZ'选项进行的。例如：

```
hbase(main):001:0> create 'testgzip', { NAME => 'cf1',COMPRESSION => 'GZ' }
```

LZO 压缩

首先参照文本表 LZO 压缩部分进行配置，然后创建 HBase 表时指定压缩选项：

```
hbase(main):001:0> create 'testlzo', { NAME => 'cf1',COMPRESSION => 'lzo' }
```

Snappy 压缩

在 core-site.xml 的 io.compression.codecs 属性对应的 value 部分加入对 Snappy 的支持：

```
<property>
<name>io.compression.codecs</name>
<value>
......
org.apache.hadoop.io.compress.SnappyCodec</value>
</property>
```

将 hadoop-snappy-***.jar 复制到 hadoop 以及 hbase 的 lib 下，然后在创建 HBase 表时指定压缩选项：

```
hbase(main):001:0> create 'testsny', { NAME => 'cf1',COMPRESSION => 'snappy' }
```

7.7.6 限制

针对 Impala 使用 HBase 表的限制，有些是 Hive 和 HBase 先天的，有些是针对 Impala 的：

（1）如果我们在 Impala 中执行 DROP TABLE 操作，对应的 HBase 表不会被删除。如果在 Hive 中执行 DROP TABLE 操作，也是同样的情况。

（2）INSERT OVERWRITE 语句对 HBase 表不可用。我们可以向基于 HBase 的 Impala 表中插入数据，也可以通过插入相同行键的记录来变相的实现 UPDATE 操作，但是无法通过 INSERT…OVERWRITE 实现对整个表的数据进行替换。而在 Hive 中，则可以通过 INSERT…OVERWRITE 来完成该功能。

（3）避免使用 CREATE TABLE LIKE 语句。我们可以执行 CREATE TABLE LIKE 语句来创建一张与另外一张表类似的、基于 HBase 的 Impala 表，但是创建的新表虽然名称与原表不同，但是新表指向的 HBase 基表完全相同，所以这样做的结果只是指定了一张原有表的别名而已。

（4）慎用 INSERT … SELECT 语句。在数据加载部分我们已经解释过，通过这种方式向基于 HBase 的 Impala 中插入的数据可能比源数据的行数要少，我们必须从业务上确认这样的结果是合理的。

7.7.7 示例

在 HBase shell 中，我们可以执行表的创建和删除操作。表创建好之后，其状态默认为“enable”，如果要删除这张表，必须先使用 DISABLE ‘table_name’将这张表的状态置为 “disable”。

1. 字符串类型行键

在 Hive shell 中执行 CREATE TABLE 语句：

```
$ hive
...
hive> CREATE EXTERNAL TABLE hbasestringids (
id string,
bool_col boolean,
tinyint_col tinyint,
smallint_col smallint,
int_col int,
bigint_col bigint,
float_col float,
double_col double,
date_string_col string,
string_col string,
timestamp_col timestamp)
```

```
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" =
":key,bools:bool_col,ints:tinyint_col,ints:smallint_col,ints:int_col,ints:\
bigint_col,floats:float_col,floats:double_col,strings:date_string_col,\
strings:string_col,strings:timestamp_col"
)
TBLPROPERTIES("hbase.table.name" = "hbasealltypessmall");
```

该示例演示了创建一张映射到 HBase 表的外部表。这张外部表在 Hive 和 Impala 中都可以进行查询，也可以进行 DROP 操作，但是删除操作仅会删除这张外部表的定义，不会将 HBase 中的基表执行真正的删除。Impala 目前不支持该语句中的 STORED BY 子句，所以只能在 Hive 的 shell 中执行。WITH SERDEPROPERTIES 子句指定第一列 id 作为行键，其余的列与 HBase 中的列族一一映射。其中 ID 列可以作为查询的条件进行单条记录的快速定位操作。而 ID 列使用 STRING 类型可以更快地对 HBase 表进行查询。

2. 非字符串类型行键

我们也可以把行键定义为其他的数据类型，如下示例所示：

```
$ hive
...
CREATE EXTERNAL TABLE hbasealltypessmall (
id int,
bool_col boolean,
tinyint_col tinyint,
smallint_col smallint,
int_col int,
bigint_col bigint,
float_col float,
double_col double,
date_string_col string,
string_col string,
timestamp_col timestamp)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
"hbase.columns.mapping" =
":key,bools:bool_col,ints:tinyint_col,ints:smallint_col,ints:int_col,ints:bigi
nt_col,floats\
:float_col,floats:double_col,strings:date_string_col,strings:string_col,string
s:timestamp_col"
)
TBLPROPERTIES("hbase.table.name" = "hbasealltypessmall");
```

虽然语法上支持这么定义，但是 Impala 强烈建议我们行键的列类型使用 STRING 类型。

第 8 章

Impala分区

默认情况下，一张表的所有的数据文件都位于同一个 HDFS 的目录中。分区是在数据加载期间将数据依据一列或者多列将数据加载到不同的物理位置，以加速对该列或者该几列查询速度的一项技术。比如，表 school_records 按照 year 列，也就是年份列进行分区，那么每一个年份将对应一个不同的数据目录，所有同一年份的数据放在同一个数据目录中。当对该表的查询使用形如 YEAR=1966，YEAR IN(1989，1999)或者 YEAR BETWEEN 1984 AND 1989 的 WHERE 条件时，Impala 只会读取年份对应的数据目录，大大减少了数据的读取量，提高了查询的效率。

8.1 分区技术适用场合

分区技术适用于以下场合：

（1）表的数据量非常庞大，单词读取整个表的时间不在我们承受的范围之内。

（2）表总是依据某些特定的列进行查询。如上述示例中的依据年份作为分区列的 school_records 表。对于查询 SELECT COUNT(*) FROM school_records WHERE year = 1985，如果表 school_records 为按年份的分区表，那么只需要读取 1985 年的相应的数据目录的数据即可；而如果该表不是分区表，那么就要对表中的全部数据进行读取，效率将大大降低。如果我们最经常使用的查询不是按年份进行查询，而是按名字，学号或者其他条件进行查询，可能按年份分区就不是一个好的选择了。

（3）分区列要有一定的区分度，也就是要包含一定数据的非重复值。如果单列的非重复值很少，比如性别列，只有两个值，男或者女，如果我们针对性别列进行查询，无论是男还是女，理论上都能匹配到一半的数据，查询效率的提升并不明显。分区列包含的非重复值越多，针对某个特定的值匹配到的数据条数越少，同时与该值对应的数据目录中的数据文件越小，对效率的提升越明显。比如，对于人口普查的数据，我们可能按年份分区，对于销售报表数据，我们需要按年和月进行分区，对于某些庞大的网站流量数据，我们需要精确到天，甚至到小时或者分钟。

（4）数据已经经过 ETL 处理。不同分区的键值最终会被分离的放在不同的数据目录中，因此将数据加载到分区表之前需要进行必要的转换和预处理。

8.2 分区表相关 SQL 语句

使用分区影响到的 Impala SQL 的语法如下：

- CREATE TABLE：我们可以指定一个 PARTITONED BY 子句创建一张指定了分区列类型和名称的分区表。分区列不能被包括在 CREATE TABLE 指定的其他列定义的列表中。
- ALTER TABLE：我们可以进行添加或者删除分区操作。对于过期的，不再需要的数据，我们可以将其对应的日期型分区删除。
- INSERT：当我们向分区表中插入数据时，我们需要指定分区列。新插入的数据存储在哪个数据目录的数据文件中，是由分区列的键值决定的。我们也可以使用 INSERT OVERWRITE 语句将一组数据显式地加载到某个特定的分区来替换整个分区的内容。

虽然 SELECT 语法的用法与表是否为分区表没有什么不同，但是针对分区表的 SELECT 查询在性能和扩展性上的影响却非常大。

8.3 分区修剪

分区修剪技术指的是一个查询可以跳过一个或者多个分区对应的数据文件的一项技术。如果我们在执行计划中发现通过分区修剪技术跳过了大量的没必要扫描的分区，那么这个查询肯定可以使用更少的资源，而且运行效率也更高。

例如：如果一张表通过 YEAR、MONTH、DAY 分区，而 WHERE 条件中包含了形如 WHERE year=2013，WHERE year<2010 或 WHERE year BETWEEN 1995 AND 1998 这样的子句，Impala 将会跳过所有的与条件中给出的不相关年份的分区对应的数据目录。如果查询条件形如 WHERE year=2013 AND month BETWEEN 1 AND 3，更为精确的条件将通过分区修剪技术忽略掉更多的与月份不相关的分区，对整个查询来说读取的数据量更小。

为了有效地确认分区修剪技术是否在一个查询中生效，我们可以通过 EXPLAIN 输出来检查确认。如下示例中有一张包含了三个分区的表，而查询只读取其中一个分区。执行计划中的“#partitions=1/3”标识着 Impala 可以通过分区修剪技术只读取三个分区中一个分区的数据就可以完成这个查询。

```
    [localhost:21000]  >  insert  into  census  partition  (year=2010)  values
('Smith'),('Jones');
    [hadoop-cs1:21000] > insert into census partition (year=2011) values
    ('Smith'),('Jones'),('Doe');
    [hadoop-cs1:21000]  >  insert  into  census  partition  (year=2012)  values
```

```
('Smith'),('Doe');
    [hadoop-cs1:21000] > select name from census where year=2010;
    +-------+
    | name |
    +-------+
    | Smith |
    | Jones |
    +-------+
    [hadoop-cs1:21000] > explain select name from census where year=2010;
    +------------------------------------------------------------------+
    | Explain String |
    +------------------------------------------------------------------+
    | PLAN FRAGMENT 0 |
    | PARTITION: UNPARTITIONED |
    | |
    | 1:EXCHANGE |
    | |
    | PLAN FRAGMENT 1 |
    | PARTITION: RANDOM |
    | |
    | STREAM DATA SINK |
    | EXCHANGE ID: 1 |
    | UNPARTITIONED |
    | |
    | 0:SCAN HDFS |
    | table=predicate_propagation.census #partitions=1/3 size=12B |
    +------------------------------------------------------------------+
```

Impala 不但能对直接按分区键值作为查询条件的语句进行分区修剪，而且也可以对 WHERE 条件中的分区列使用传递属性进行分区修剪。这项技术被称为谓词推进，在 Impala 1.2.2 或者更高版本中可以使用。在如下的示例中，调查表使用每 10 年作为一个分区的时间间隔。示例中的查询使用的形式不是分区列等于一个常量值，但是 Impala 通过传递属性可以分析出分区列等于某个常量是查询的必要条件，并进行分区修剪。

```
    [hadoop-cs1:21000] > drop table census;
    [hadoop-cs1:21000] > create table census (name string, census_year int) partitioned
by
    (year int);
    [hadoop-cs1:21000] > insert into census partition (year=2010) values
```

```
('Smith',2010),('Jones',2010);
[hadoop-cs1:21000] > insert into census partition (year=2011) values
('Smith',2020),('Jones',2020),('Doe',2020);
[hadoop-cs1:21000] > insert into census partition (year=2012) values
('Smith',2020),('Doe',2020);
[hadoop-cs1:21000] > select name from census where year = census_year and
census_year=2010;
+--------+
| name |
+--------+
| Smith |
| Jones |
+--------+
[hadoop-cs1:21000] > explain select name from census where year = census_year and
census_year=2010;
+------------------------------------------------------------------+
| Explain String |
+------------------------------------------------------------------+
| PLAN FRAGMENT 0 |
| PARTITION: UNPARTITIONED |
| |
| 1:EXCHANGE |
| |
| PLAN FRAGMENT 1 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 1 |
| UNPARTITIONED |
| |
| 0:SCAN HDFS |
| table=predicate_propagation.census #partitions=1/3 size=22B |
| predicates: census_year = 2010, year = census_year |
+------------------------------------------------------------------+
```

如果要了解该语句执行的更多细节，请在执行该语句后运行 PROFILE 命令进一步检查。

如果视图的基表是一张分区表，那么分区修剪依据的唯一条件就是该视图定义的原始语句本身。Impala 不会对视图查询指定的条件进行进一步的分区修剪操作。

8.4 分区键列

一般情况下，选择最频繁使用的，能够对大量数据进行过滤的列作为分区列。我们最经常使用的分区列包括和日期时间相关的年份、月份、日期等，另外，区域的物理位置也可以作为分区列使用。

（1）对于基于时间的数据，我们需要将需要使用的、作为分区键的部分单独抽取出来。因为 Impala 不能基于 TIMESTAMP 列分区。

（2）分区列的数据类型对真正存储在 HDFS 上的数据文件的大小影响不大。因为分区列的值不是真正地存储到数据文件中，而是以字符串的形式作为 HDFS 数据目录名称的一部分。

（3）我们要时刻谨记 Impala 查询的是存储在 HDFS 上的数据。数据文件至少是几十兆或者更大，对 HDFS 的 IO 操作才更有优势。对于 Parquet 表，数据块大小为 1GB。因此，我们要尽量避免由于选择的分区键包含太多的值，而使每个分区值对应的数据文件个数过多，单个数据文件过小。比如，如果我们每天产生 1GB 的数据，那么我们可以按年，月，日分区。如果我们每分钟产生 5GB 的数据，那么我们可以按年、月、日、小时、分钟作为分区列。如果我们的数据包含地理位置信息，我们可以用省份、地市、县、或者地区的邮政编码作为分区列。

8.5 使用不同的文件格式

分区表对文件格式的要求具有很大的灵活性，我们甚至可以为不同的分区使用不同的文件格式。例如，我们的表使用纯文本的文件格式，随后我们使用 RCFile 作为新的文件格式接收数据，或者也可以再使用 Parquet 作为新的文件格式，而所有的数据可以在同一张表中进行查询。但是这样做的前提是，我们必须保证不同的分区使用了不同的文件格式。

如下示例中展示了如何将 text 类型转换为 Parquet 类型：

```
    [hadoop-cs1:21000] > create table census (name string) partitioned by (year
smallint);
    [hadoop-cs1:21000] > alter table census add partition (year=2012); -- Text format;
    [hadoop-cs1:21000] > alter table census add partition (year=2013); -- Text format
switches
    to Parquet before data loaded;
    [hadoop-cs1:21000] > alter table census partition (year=2013) set fileformat
parquet;
    [hadoop-cs1:21000] > insert into census partition (year=2012) values
    ('Smith'),('Jones'),('Lee'),('Singh');
```

```
[hadoop-cs1:21000] > insert into census partition (year=2013) values
('Flores'),('Bogomolov'),('Cooper'),('Appiah');
```

这个示例中 year=2012 分区使用了 text 文件格式，year=2013 分区使用了 Parquet 文件格式。当然，生产环境中一般不会使用 INSERT … VALUES 这种方式插入数据，而是使用 INSERT … SELECT 或者 LOAD DATA 方式向表中加载数据。

对于其他的数据文件类型，我们无法直接使用 Impala 进行创建。但是我们可以通过在 Hive 中执行 ALTER TABLE … SET FILEFORMAT、INSERT 或者 LOAD DATA 方式将数据加载进特定的数据文件格式中。最后，我们在 Impala 中使用 REFRESH table_name 来更新该表的元数据信息，让 Impala 可以识别到新的数据文件和数据。

第 9 章

Impala性能优化

本章将主要介绍影响 Impala 性能的主要因素，以及监控，调整，压力测试的整个过程。

另外，本章还描述了如何最大限度地提高 Impala 的可扩展性问题。可扩展性和性能密不可分，即要保证随着系统负载的不断增加，系统仍然可以保持良好的性能。比如：我们通过对单个的查询优化减小的磁盘 IO，那也就意味着同时其他查询可以使用更多的磁盘 IO，也就变相的提高了系统的可扩展性。跟磁盘 IO 一样的道理，如果我们对一个查询的优化可以减少内存的使用，那么说明其他查询在同一时间可以使用更多的内存资源，也是变相的提高了系统的可扩展性。在某些情况下，通过优化技术提高系统的可扩展性比提升性能本身更加重要。

下面给出性能优化的几个重要术语的定义：

- 分区：这项技术在物理上基于分区的键值将数据分开存放，这样在基于键值列查询时，只需对相应的键值列数据进行操作就可以返回结果。
- 连接查询：连接优化是我们在 SQL 级别能够进行的最主要的优化之一。如果从 SQL 级别能够把查询优化到我们容许的时间内，就不必进行像改变数据文件格式或者改变硬件配置的其他优化。
- 表统计信息：使用 COMPUTE STATS 语句搜集表和列统计信息可以帮助 Impala 自动优化连接查询。
- 性能测试：在进行基准测试之前，我们要通过进行测试确保 Impala 使用的配置达到了最佳性能。
- 基准测试：我们使用的初始化配置和样本数据往往不适合做性能测试。
- 控制资源使用：一般情况下 Impala 使用的内存越多，查询的性能越好。然而在一个集群中，我们可能除了 Impala，还有其他不同的 Hadoop 组件，我们要保证所有的组件都能申请到足够的内存资源。

9.1 最佳实践

这里列出的指导原则和最佳实践可以指导我们规划、测试和性能优化。

（1）为数据选择合适的文件格式。

通常情况下，对于海量数据（每张表或者每个分区至少要有几个 GB）的存储，Parquet 文件

格式具有很大的优势，因为它按列存储，单次 IO 可以请求更多数据，另外它也有很好的压缩算法对二进制文件进行压缩。

（2）避免在数据处理过程中产生很多小文件。

使用 INSERT…SELECT 在表表之间拷贝数据。避免对海量数据或者影响性能的关键表使用 INSERT…VALUES 插入数据，因为每条这样的 INSERT 语句都会产生单个的小文件。

如果在数据处理过程中产生了上千个小文件，我们需要使用 INSERT…SELECT 来将数据复制到另外一张表，在复制的过程中也解决了小文件过多的问题。

（3）合适的使用分区技术。比如，如果我们有一个包含上千个分区的 Parquet 表，每个分区的数据都小于 1GB，那么我们就要考虑以更大的粒度来作为分区的提交，比如如果原来的分区键值是年月日，那么现在我们就可以考虑只使用年和月。只有分区的粒度使数据文件的大小合适，才能充分利用 HDFS 的 IO 批处理性能和 Impala 的分布式查询。

（4）使用 COMPUTE STATS 搜集连接查询中海量数据表或者影响性能的关键表的统计信息。

（5）最小化向客户端传输结果的开销。可以考虑使用如下技术：

- 聚集：如果我们需要计算满足某个条件的记录行数，求匹配到的行数中某些列的和，最大值、最小值等，不要将整个结果集发送到客户端由客户端应用来处理这些数据，而是可以调用像 COUNT()、SUM()、MAX()等聚集函数来处理。如果将未聚集过的数据整个发送给客户端，单单将数据传送到客户端这一个动作就需要消耗很大的网络开销。
- 过滤：使用不同的谓词条件尽可能的缩小结果集的大小，而不是把整个结果集发送到应用端，由应用来处理过滤的逻辑。
- LIMIT 子句：如果我们只需要查看很少的样本数据，或者查看使用 ORDER BY 之后产生的最大值或者最小值，可以使用 LIMIT 子句来最大限度的减小结果集的大小。
- 避免对结果集使用美化输出：当我们通过 impala-shell 获取数据时，可以指定-B 和 --output_delimiter 选项输出原始的结果集，而不需要 Impala 对输出的格式进行美化，或者直接将结果集重定向到文件中。上述的情况也可以考虑直接对查询的语句使用 INSERT …SELECT 将结果集直接写到 HDFS 上。

（6）在实际运行一个查询之前，先使用 EXPLAIN 查看它的执行计划是否将以高效合理的方式运行。

（7）在运行一个查询之后，使用 PROFILE 命令从底层确认 IO，内存消耗，网络带宽占用，CPU 使用率等信息是否在我们期望的范围之内。

9.2 连接查询优化

涉及到连接的查询往往比对单表查询更需要优化操作。连接查询产生的最大结果集是参与关

联的所有表的记录数的乘积。当我们对百万级别的表和十亿级别的表关联时，忽略对结果集的提前过滤的操作，或者其他可能拖慢查询速度的因素都有可能导致查询不能在预期的时间内完成，甚至导致查询被终止。

优化连接最简单的方式就是使用 COMPUTE STATS 命令搜集所有参与关联表的统计信息，让 Impala 根据每个表的大小、列的非重复值个数等相关信息自动优化查询。COMPUTE STATS 和连接查询的优化都是自 Impala 1.2.2 开始提供的功能。为了保证统计信息的准确性，我们需要在对表 INSERT、LOAD DATA 或者添加分区等操作之后及时执行 COMPUTE STATS 命令搜集统计信息。

如果参与关联的表的统计信息不可用，而且 Impala 自动选择的连接顺序效率很低，我们可以在 SELECT 关键字后使用 STRAIGHT_JOIN 关键字手动指定连接的顺序。指定了该关键字之后，Impala 将使用表在查询中出现的先后顺序作为关联顺序进行处理。

当我们使用了 STRAIGHT_JOIN 关键字之后，我们必须保证查询中表的先后顺序，也就是表的连接顺序，而不能依赖于 Impala 优化器。对于自动优化的查询，优化器能够计算出连接的每个阶段结果集的大小。而对于手动指定连接顺序的查询，我们可能需要根据情况对连接顺序进行微调：

（1）指定最大的表作为第一张表。在查询执行的这个阶段，只是把数据从 Impala 节点的磁盘上读出来，对内存的消耗并不严重。

（2）指定最小的表作为下一张表。后续的第二张表、第三张表等都需要经过网络传输。如果我们想保持后续连接查询的每个阶段结果集的大小，最有可能的做法就是先和一张最小的表关联，这样生成的结果集也是最小的。

（3）接着指定剩下的表中最小的表作为下一张表，以此类推。比如：如果有四张表分别为 BIG、MEDIUM、SMALL、TINY，那么按照上述的说明连接的顺序应该是：BIG、TINY、SMALL、MEDIUM。

Impala 查询优化器根据表的绝对大小和相对大小为连接查询选择不同的关联技术，它提供了两种连接方式：

- 默认的连接方式是 Broadcast 连接，当右手表比左手表小时，它的内容会被发送到所有执行查询的节点上。
- 另外一种连接方式是 partitioned 连接，它使用于大小差不太多的大表关联。使用此种方式关联，为了保证关联操作可以并行执行，每个表的一部分数据都会被发送到不同节点上，最后各节点分别对传送过来的数据并行处理。

具体 Impala 优化器选择哪种连接方式，完全依赖于通过 COMPUTE STATS 搜集的表的统计信息。

为了确认表的连接策略，我们可以对一个特定的查询执行 EXPLAIN 语句。如果通过基准测试我们可以确认一种连接方式比另一种连接方式效率更高，也可以通过 Hint 的方式手动指定需要的连接方式。

1. 当统计信息不可用时如何关联

如果参与关联的某些表的统计信息还是可用的，Impala 会根据存在统计信息的表重新生成连接顺序。有统计信息的表会被放置在连接顺序的最左端，并根据表的基数和规模降序排列。而没有统计信息的表被作为空表对待，总是放在连接顺序的最右端。

2. 使用 STRAIGHT_JOIN 覆盖连接顺序

如果关联查询由于统计信息过期或者数据分布等问题导致效率低下，我们可以通过指定 STRAIGHT_JOIN 关键字改变连接顺序。使用该关键字后，关联查询将不会使用 Impala 查询优化器自动生成的连接顺序，而是使用查询中表出现的先后顺序作为关联的顺序。

在如下示例中，表 BIG 经过过滤实际上产生了一个非常小的结果集，而 Impala 仍然把它作为最大的表对待放在连接顺序的最左侧。为了改变优化器错误的判断，我们使用 STRAIGHT_JOIN 改变连接的顺序，把 BIG 表放到了连接顺序的最右侧：

```
select straight_join x from medium join small join (select * from big where c1 <
10)
as big
where medium.id = small.id and small.id = big.id;
```

3. 连接顺序优化示例

示例中的表分别有 10 亿行、2 亿行、100 万行。这里的表都不是非区表，而且使用 Parquet 格式存储。比较小的表的数据都是最大的表的列的子集，它们都使用唯一列 ID 进行关联。

```
[hadoop-cs1:21000] > create table big stored as parquet as select * from raw_data;
+----------------------------+
| summary |
+----------------------------+
| Inserted 1000000000 row(s) |
+----------------------------+
Returned 1 row(s) in 671.56s
[hadoop-cs1:21000] > desc big;
+-----------+---------+---------+
| name | type | comment |
+-----------+---------+---------+
| id | int | |
| val | int | |
| zfill | string | |
| name | string | |
| assertion | boolean | |
+-----------+---------+---------+
```

```
Returned 5 row(s) in 0.01s
[hadoop-cs1:21000] > create table medium stored as parquet as select * from big
limit
200 * floor(1e6);
+---------------------------+
| summary |
+---------------------------+
| Inserted 200000000 row(s) |
+---------------------------+
Returned 1 row(s) in 138.31s
[hadoop-cs1:21000] > create table small stored as parquet as select id,val,name
from
big where assertion = true limit 1 * floor(1e6);
+-------------------------+
| summary |
+-------------------------+
| Inserted 1000000 row(s) |
+-------------------------+
Returned 1 row(s) in 6.32s
```

在实际运行查询之前使用 EXPLAIN 查看连接的信息。启用执行计划的详细输出，我们可以看到更多性能相关的输出信息：需要着重提示的信息以黑体显示。输出信息提示我们参与关联的表没有统计信息，Impala 不能为每个执行阶段估计出结果集的大小，它坚持使用 BROADCAST 的方式向每个节点发送一个表的完整副本。

```
[hadoop-cs1:21000] > set explain_level=verbose;
EXPLAIN_LEVEL set to verbose
[hadoop-cs1:21000] > explain select count(*) from big join medium where big.id =
medium.id;
+----------------------------------------------------------+
| Explain String
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=2.10GB VCores=2 |
| |
| PLAN FRAGMENT 0 |
| PARTITION: UNPARTITIONED |
| |
| 6:AGGREGATE (merge finalize) |
| | output: SUM(COUNT(*)) |
```

```
| | cardinality: 1 |
| | per-host memory: unavailable |
| | tuple ids: 2 |
| | |
| 5:EXCHANGE |
| cardinality: 1 |
| per-host memory: unavailable |
| tuple ids: 2 |
| |
| PLAN FRAGMENT 1 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 5 |
| UNPARTITIONED |
| |
| 3:AGGREGATE |
| | output: COUNT(*) |
| | cardinality: 1 |
| | per-host memory: 10.00MB |
| | tuple ids: 2 |
| | |
| 2:HASH JOIN |
| | join op: INNER JOIN (BROADCAST) |
| | hash predicates: |
| | big.id = medium.id |
| | cardinality: unavailable |
| | per-host memory: 2.00GB |
| | tuple ids: 0 1 |
| | |
| |----4:EXCHANGE |
| | cardinality: unavailable |
| | per-host memory: 0B |
| | tuple ids: 1 |
| | |
| 0:SCAN HDFS |
| table=join_order.big #partitions=1/1 size=23.12GB |
| table stats: unavailable |
| column stats: unavailable |
```

```
| cardinality: unavailable |
| per-host memory: 88.00MB |
| tuple ids: 0 |
| |
| PLAN FRAGMENT 2 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 4 |
| UNPARTITIONED |
| |
| 1:SCAN HDFS |
| table=join_order.medium #partitions=1/1 size=4.62GB |
| table stats: unavailable |
| column stats: unavailable |
| cardinality: unavailable |
| per-host memory: 88.00MB |
| tuple ids: 1 |
+------------------------------------------------------------+
Returned 64 row(s) in 0.04s
```

为每张表执行 COMPUTE STATS 搜集统计信息：

```
[hadoop-cs1:21000] > compute stats small;
+-------------------------------------------+
| summary |
+-------------------------------------------+
| Updated 1 partition(s) and 3 column(s). |
+-------------------------------------------+
Returned 1 row(s) in 4.26s
[hadoop-cs1:21000] > compute stats medium;
+-------------------------------------------+
| summary |
+-------------------------------------------+
| Updated 1 partition(s) and 5 column(s). |
+-------------------------------------------+
Returned 1 row(s) in 42.11s
[hadoop-cs1:21000] > compute stats big;
+-------------------------------------------+
| summary |
```

```
+----------------------------------------+
| Updated 1 partition(s) and 5 column(s). |
+----------------------------------------+
Returned 1 row(s) in 165.44s
```

搜集完统计信息之后，Impala 可以依据统计信息选择更有效率的连接顺序，而选择 BROADCAST 还是 PARTITIONED 连接方式仍然是依据表的大小和行数的差别来确定。

```
[hadoop-cs1:21000] > explain select count(*) from medium join big where big.id =
medium.id;
Query: explain select count(*) from medium join big where big.id = medium.id
+----------------------------------------------------------+
| Explain String |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=937.23MB VCores=2 |
| |
| PLAN FRAGMENT 0 |
| PARTITION: UNPARTITIONED |
| |
| 6:AGGREGATE (merge finalize) |
| | output: SUM(COUNT(*)) |
| | cardinality: 1 |
| | per-host memory: unavailable |
| | tuple ids: 2 |
| | |
| 5:EXCHANGE |
| cardinality: 1 |
| per-host memory: unavailable |
| tuple ids: 2 |
| |
| PLAN FRAGMENT 1 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 5 |
| UNPARTITIONED |
| |
| 3:AGGREGATE |
| | output: COUNT(*) |
| | cardinality: 1 |
```

```
| | per-host memory: 10.00MB |
| | tuple ids: 2 |
| | |
| 2:HASH JOIN |
| | join op: INNER JOIN (BROADCAST) |
| | hash predicates: |
| | big.id = medium.id |
| | cardinality: 1443004441 |
| | per-host memory: 839.23MB |
| | tuple ids: 1 0 |
| | |
| |----4:EXCHANGE |
| | cardinality: 200000000 |
| | per-host memory: 0B |
| | tuple ids: 0 |
| |
| 1:SCAN HDFS |
| table=join_order.big #partitions=1/1 size=23.12GB |
| table stats: 1000000000 rows total |
| column stats: all |
| cardinality: 1000000000 |
| per-host memory: 88.00MB |
| tuple ids: 1 |
| |
| PLAN FRAGMENT 2 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 4 |
| UNPARTITIONED |
| |
| 0:SCAN HDFS |
| table=join_order.medium #partitions=1/1 size=4.62GB |
| table stats: 200000000 rows total |
| column stats: all |
| cardinality: 200000000 |
| per-host memory: 88.00MB |
| tuple ids: 0 |
+------------------------------------------------------------+
```

```
Returned 64 row(s) in 0.04s


[hadoop-cs1:21000] > explain select count(*) from small join big where big.id =
small.id;
Query: explain select count(*) from small join big where big.id = small.id
+------------------------------------------------------------+
| Explain String |
+------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=101.15MB VCores=2 |
| |
| PLAN FRAGMENT 0 |
| PARTITION: UNPARTITIONED |
| |
| 6:AGGREGATE (merge finalize) |
| | output: SUM(COUNT(*)) |
| | cardinality: 1 |
| | per-host memory: unavailable |
| | tuple ids: 2 |
| | |
| 5:EXCHANGE |
| cardinality: 1 |
| per-host memory: unavailable |
| tuple ids: 2 |
| |
| PLAN FRAGMENT 1 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 5 |
| UNPARTITIONED |
| |
| 3:AGGREGATE |
| | output: COUNT(*) |
| | cardinality: 1 |
| | per-host memory: 10.00MB |
| | tuple ids: 2 |
| | |
| 2:HASH JOIN |
```

```
| | join op: INNER JOIN (BROADCAST) |
| | hash predicates: |
| | big.id = small.id |
| | cardinality: 1000000000 |
| | per-host memory: 3.15MB |
| | tuple ids: 1 0 |
| | |
| |----4:EXCHANGE |
| | cardinality: 1000000 |
| | per-host memory: 0B |
| | tuple ids: 0 |
| | |
| 1:SCAN HDFS |
| table=join_order.big #partitions=1/1 size=23.12GB |
| table stats: 1000000000 rows total |
| column stats: all |
| cardinality: 1000000000 |
| per-host memory: 88.00MB |
| tuple ids: 1 |
| |
| PLAN FRAGMENT 2 |
| PARTITION: RANDOM |
| |
| STREAM DATA SINK |
| EXCHANGE ID: 4 |
| UNPARTITIONED |
| |
| 0:SCAN HDFS |
| table=join_order.small #partitions=1/1 size=17.93MB |
| table stats: 1000000 rows total |
| column stats: all |
| cardinality: 1000000 |
| per-host memory: 32.00MB |
| tuple ids: 0 |
+------------------------------------------------------------+
Returned 64 row(s) in 0.03s
```

而实际执行查询时发现无论表的连接顺序如何，实际的执行时间都相差不多。因为样本数据中的 ID 列和 VAL 列都包含很多重复值：

```
[hadoop-cs1:21000] > select count(*) from big join small on (big.id = small.id);
Query: select count(*) from big join small on (big.id = small.id)
+----------+
| count(*) |
+----------+
| 1000000 |
+----------+
Returned 1 row(s) in 21.68s
[hadoop-cs1:21000] > select count(*) from small join big on (big.id = small.id);
Query: select count(*) from small join big on (big.id = small.id)
+----------+
| count(*) |
+----------+
| 1000000 |
+----------+
Returned 1 row(s) in 20.45s
[hadoop-cs1:21000] > select count(*) from big join small on (big.val = small.val);
+------------+
| count(*) |
+------------+
| 2000948962 |
+------------+
Returned 1 row(s) in 108.85s
[hadoop-cs1:21000] > select count(*) from small join big on (big.val = small.val);
+------------+
| count(*) |
+------------+
| 2000948962 |
+------------+
Returned 1 row(s) in 100.76s
```

9.3 使用统计信息

当统计信息可用时，Impala 可以依据表中数据的规模、值的分布等信息对复杂查询或者多表查询进行很好的优化。本部分将介绍 Impala 使用的统计信息的类别，如何产生并保持最新的统计信息。

在早期版本中，Impala 对统计信息的搜集依赖于 Hive 的统计信息搜集机制，需要使用 ANALYZE TABLE 语句运行一个 MapReduce 作业。从用户友好和可靠性考虑，Impala 实现了自己的 COMPUTE STATS、SHOW TABLE STATS、SHOW COLUMN STATS 语句。

1. 表统计信息

当元数据库中有表或者分区的统计信息元数据时，Impala 查询优化器会充分利用这些信息，并结合列统计信息为查询语句自动生成优化的执行路径。

搜集表统计信息有如下方式：

（1）在 Impala 中执行 COMPUTE STATS 语句。该语句自 Impala 1.2.2 起开始支持，是最好的搜集统计信息的方式：

- 使用这个语句将自动对表、分区及列统计信息进行搜集，使用一条语句即可完成。
- 它不依赖于 Hive 的配置，元数据配置及持有统计信息的元数据库是否是单独等等限制。
- 可以对已搜集过统计信息的表进行增量搜集。比如，我们为表添加了一个分区，或者插入了新的数据，我们可以使用 ALTER TABLE 语句只搜集更新的属性信息，而不是对整个表进行重新搜集:

```
alter table analysis_data set tblproperties('numRows'='new_value');
```

（2）当 hive.stats.autogather 设置为启用时，在 Hive 中通过 INSERT OVERWRITE 语句加载数据。

（3）在 Hive 中使用 ANALYZE TABLE 语句对整个表或某个分区搜集统计信息。

```
   ANALYZE  TABLE  tablename  [PARTITION(partcol1[=val1],  partcol2[=val2],  ...)]
COMPUTE
   STATISTICS [NOSCAN];
```

- 为一个非分区表搜集统计信息:

```
ANALYZE TABLE customer COMPUTE STATISTICS;
```

- 为一个分区的所有分区搜集统计信息:

```
ANALYZE TABLE store PARTITION(s_state, s_county) COMPUTE STATISTICS;
```

- 为一个分区表的特定分区搜集统计信息:

```
ANALYZE TABLE store PARTITION(s_state='CA', s_county) COMPUTE STATISTICS;
```

为了检查确认统计信息是否可用，或者像查看更多相关细节，可以使用 SHOW TABLE STAT table_name 语句。

如果使用基于 Hive 的方式搜集统计信息，在 Hive 侧需要进行正确的配置。Cloudera 推荐使用 Impala 的 COMPUTE STATS 语句搜集统计信息，使用这种方式可以避免搜集过程中潜在的配

置及可扩展性的问题。

2. 列统计信息

Impala 查询优化器可以使用元数据库中单独的列统计信息。这项技术对于跨表的连接查询计算连接后返回的数据集的大小非常有效。目前，Impala 不能自己创建这些元数据信息，但是在 Hive shell 中使用 ANALYZE TABLE 语句可以搜集这些统计信息。

如果我们要检查确认表的列统计信息是否可用，可以使用 SHOW COLUMN STATS table_name 语句，或者使用 EXPLAIN 语句。

3. 通过 ALTER TABLE 手动设置统计信息

对于表或者某个分区来说 最重要的统计信息之一就是表或者分区的行数 COMPUTE STATS 语句既可以搜集所有列的统计信息，也可以搜集表的统计信息。在生产环境中，如果表添加了分区或者插入了数据就执行 COMPUTE STATS 来搜集统计信息并不是唯一的做法 但是如果我们可以大致判断数据量变化足以改变执行计划，我们可以通过 ALTER TABLE 语句手动设置具体的行数的统计信息。

```
create table analysis_data stored as parquet as select * from raw_data;
Inserted 1000000000 rows in 181.98s
compute stats analysis_data;
insert into analysis_data select * from smaller_table_we_forgot_before;
Inserted 1000000 rows in 15.32s
```

我们可以手动的更新表的总行数的统计信息：

```
alter table analysis_data set tblproperties('numRows'='1001000000');
```

对于一个分区表，我们可以不但可以设置整表的行数值，也可以设置每个分区的行数：

```
alter table partitioned_data partition(year=2009, month=4) set tblproperties
('numRows'='30000');
alter table partitioned_data set tblproperties ('numRows'='1030000');
```

在实践中，大多数情况下使用 COMPUTE STATS 语句已经可以有效地搜集更新这些元数据的变化。但是如果我们在只需要对 numRows 进行调整以获得更好的执行计划时，我们可以考虑使用 ALTER TABLE 手动设置这些统计信息。比如，如果我们想改变多表关联的连接顺序时，就可以使用这项技术。

4. 统计信息使用示例

如下的示例我们将通过使用 SHOW TABLE STATS、SHOW COLUMN STATS、ALTER TABLE 以及 SELECT 和 INSERT 语句来从不同的方面描述 Impala 是如何使用统计信息来进行优化操作的。

示例中使用的表是 TPC-DS 基准测试中使用的。在没有搜集统计信息之前，大多数的数字字段显示的统计信息值为-1，表示未知统计信息。但是根据仅有的信息，我们仍然可以清楚地看到像数据文件的个数，数据文件的大小，数据类型的最大长度或者平均长度等信息。

```
[hadoop-cs1:21000] > show table stats store;
+-------+--------+--------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+--------+--------+
| -1 | 1 | 3.08KB | TEXT |
+-------+--------+--------+--------+
Returned 1 row(s) in 0.03s
[hadoop-cs1:21000] > show column stats store;
+--------------------+-----------+------------------+--------+----------+-----
-----+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------------------+-----------+------------------+--------+----------+-----
-----+
| s_store_sk | INT | -1 | -1 | 4 | 4 |
| s_store_id | STRING | -1 | -1 | -1 | -1 |
| s_rec_start_date | TIMESTAMP | -1 | -1 | 16 | 16 |
| s_rec_end_date | TIMESTAMP | -1 | -1 | 16 | 16 |
| s_closed_date_sk | INT | -1 | -1 | 4 | 4 |
| s_store_name | STRING | -1 | -1 | -1 | -1 |
| s_number_employees | INT | -1 | -1 | 4 | 4 |
| s_floor_space | INT | -1 | -1 | 4 | 4 |
| s_hours | STRING | -1 | -1 | -1 | -1 |
| s_manager | STRING | -1 | -1 | -1 | -1 |
| s_market_id | INT | -1 | -1 | 4 | 4 |
| s_geography_class | STRING | -1 | -1 | -1 | -1 |
| s_market_desc | STRING | -1 | -1 | -1 | -1 |
| s_market_manager | STRING | -1 | -1 | -1 | -1 |
| s_division_id | INT | -1 | -1 | 4 | 4 |
| s_division_name | STRING | -1 | -1 | -1 | -1 |
| s_company_id | INT | -1 | -1 | 4 | 4 |
| s_company_name | STRING | -1 | -1 | -1 | -1 |
| s_street_number | STRING | -1 | -1 | -1 | -1 |
| s_street_name | STRING | -1 | -1 | -1 | -1 |
| s_street_type | STRING | -1 | -1 | -1 | -1 |
| s_suite_number | STRING | -1 | -1 | -1 | -1 |
```

```
| s_city | STRING | -1 | -1 | -1 | -1 |
| s_county | STRING | -1 | -1 | -1 | -1 |
| s_state | STRING | -1 | -1 | -1 | -1 |
| s_zip | STRING | -1 | -1 | -1 | -1 |
| s_country | STRING | -1 | -1 | -1 | -1 |
| s_gmt_offset | FLOAT | -1 | -1 | 4 | 4 |
| s_tax_precentage | FLOAT | -1 | -1 | 4 | 4 |
+--------------------+-----------+-------------------+--------+----------+-----
-----+
Returned 29 row(s) in 0.04s
```

使用 Hive 的 ANALYZE TALBE 可以为我们指定的某列搜集列统计信息，而使用 Impala 的 COMPUTE STATS 可以搜集所有列的统计信息，因为它对整个表进行读操作，可以高效地统计出所有列的统计信息。示例中显示的是运行了 COMPUTE STATS 之后，表和列统计信息的变化情况：

```
[hadoop-cs1:21000] > compute stats store;
+-----------------------------------------+
| summary |
+-----------------------------------------+
| Updated 1 partition(s) and 29 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 1.88s
[hadoop-cs1:21000] > show table stats store;
+--------+--------+--------+--------+
| #Rows | #Files | Size | Format |
+--------+--------+--------+--------+
| 12 | 1 | 3.08KB | TEXT |
+--------+--------+--------+--------+
Returned 1 row(s) in 0.02s
[hadoop-cs1:21000] > show column stats store;
+--------------------+-----------+-------------------+--------+----------+-----
--------------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size
|
+--------------------+-----------+-------------------+--------+----------+-----
--------------+
| s_store_sk | INT | 12 | 0 | 4 | 4
|
| s_store_id | STRING | 6 | 0 | 16 | 16
|
```

```
| s_rec_start_date | TIMESTAMP | 4 | 0 | 16 | 16
|
| s_rec_end_date | TIMESTAMP | 3 | 6 | 16 | 16
|
| s_closed_date_sk | INT | 3 | 9 | 4 | 4
|
| s_store_name | STRING | 8 | 0 | 5 | 4.25
|
| s_number_employees | INT | 9 | 0 | 4 | 4
|
| s_floor_space | INT | 10 | 0 | 4 | 4
|
| s_hours | STRING | 2 | 0 | 8 |
7.083300113677979 |
| s_manager | STRING | 7 | 0 | 15 | 12
|
| s_market_id | INT | 7 | 0 | 4 | 4
|
| s_geography_class | STRING | 1 | 0 | 7 | 7
|
| s_market_desc | STRING | 10 | 0 | 94 | 55.5
|
| s_market_manager | STRING | 7 | 0 | 16 | 14
|
| s_division_id | INT | 1 | 0 | 4 | 4
|
| s_division_name | STRING | 1 | 0 | 7 | 7
|
| s_company_id | INT | 1 | 0 | 4 | 4
|
| s_company_name | STRING | 1 | 0 | 7 | 7
|
| s_street_number | STRING | 9 | 0 | 3 |
2.833300113677979 |
| s_street_name | STRING | 12 | 0 | 11 |
6.583300113677979 |
| s_street_type | STRING | 8 | 0 | 9 |
4.833300113677979 |
| s_suite_number | STRING | 11 | 0 | 9 | 8.25
```

```
|
| s_city | STRING | 2 | 0 | 8 | 6.5
|
| s_county | STRING | 1 | 0 | 17 | 17
|
| s_state | STRING | 1 | 0 | 2 | 2
|
| s_zip | STRING | 2 | 0 | 5 | 5
|
| s_country | STRING | 1 | 0 | 13 | 13
|
| s_gmt_offset | FLOAT | 1 | 0 | 4 | 4
|
| s_tax_precentage | FLOAT | 5 | 0 | 4 | 4
|
+--------------------+-----------+------------------+--------+----------+-----
--------------+
Returned 29 row(s) in 0.04s
```

下面的示例展示如何对分区表搜集统计信息。示例中我们使用一张按年份分区的表中的一个 STRING 字段来保存人口普查数据信息。表统计信息包括对每个分区的统计信息和整表的统计信息。

```
localhost:21000] > describe census;
+------+----------+---------+
| name | type | comment |
+------+----------+---------+
| name | string | |
| year | smallint | |
+------+----------+---------+
Returned 2 row(s) in 0.02s
[hadoop-cs1:21000] > show table stats census;
+-------+-------+--------+------+---------+
| year | #Rows | #Files | Size | Format |
+-------+-------+--------+------+---------+
| 2000 | -1 | 0 | 0B | TEXT |
| 2004 | -1 | 0 | 0B | TEXT |
| 2008 | -1 | 0 | 0B | TEXT |
| 2010 | -1 | 0 | 0B | TEXT |
| 2011 | 0 | 1 | 22B | TEXT |
```

```
| 2012 | -1 | 1 | 22B | TEXT |
| 2013 | -1 | 1 | 231B | PARQUET |
| Total | 0 | 3 | 275B | |
+-------+-------+--------+------+---------+
Returned 8 row(s) in 0.02s
[hadoop-cs1:21000] > show column stats census;
+--------+----------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+----------+-----------------+--------+----------+----------+
| name | STRING | -1 | -1 | -1 | -1 |
| year | SMALLINT | 7 | 0 | 2 | 2 |
+--------+----------+-----------------+--------+----------+----------+
Returned 2 row(s) in 0.02s
```

在使用 COMPUTE STATS 搜集统计信息之后：

```
[hadoop-cs1:21000] > compute stats census;
+-----------------------------------------+
| summary |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 2.16s
[hadoop-cs1:21000] > show table stats census;
+-------+-------+--------+------+---------+
| year | #Rows | #Files | Size | Format |
+-------+-------+--------+------+---------+
| 2000 | -1 | 0 | 0B | TEXT |
| 2004 | -1 | 0 | 0B | TEXT |
| 2008 | -1 | 0 | 0B | TEXT |
| 2010 | -1 | 0 | 0B | TEXT |
| 2011 | 4 | 1 | 22B | TEXT |
| 2012 | 4 | 1 | 22B | TEXT |
| 2013 | 1 | 1 | 231B | PARQUET |
| Total | 9 | 3 | 275B | |
+-------+-------+--------+------+---------+
Returned 8 row(s) in 0.02s
[hadoop-cs1:21000] > show column stats census;
+--------+----------+-----------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
```

```
+--------+----------+-----------------+--------+---------+----------+
| name | STRING | 4 | 1 | 5 | 4.5 |
| year | SMALLINT | 7 | 0 | 2 | 2 |
+--------+----------+-----------------+--------+---------+----------+
Returned 2 row(s) in 0.02s
```

如上示例展示了搜集统计信息前后的一些变化。我们可以在搜集统计信息前后查看 EXPLAIN 的输出信息，确认一个查询是如何执行的。检查搜集统计信息前后的 PROFILE 信息，确认查询时间变化、吞吐量变化，来校验对整体系统性能的影响。

9.4 基准测试

和其他的 Hadoop 组件类似，Impala 是用来做大数据量分布式查询的，因此基准测试必须使用生产环境的配置信息和真实的数据。要发挥分布式查询的优势，我们必须使用多个节点的集群环境。要发挥对海量数据查询的优势，我们必须使用 TB 级的表进行测试。

当我们运行的查询返回很多行数据时，用于结果集的格式化美化输出可能占用大量的 CPU 时间，导致最终集群用于实际的查询的时间不准确。为了避免这个问题的产生，我们可以使用 impala-shell 启动选项-B 关闭格式化美化输出，或者使用-o 选项将查询结果重定向到文件中。

9.5 控制资源使用

在生产环境中为了平衡单个查询的性能和集群的吞吐量，我们需要最集群的资源进行限制，比如一个或者一组查询使用的内存、CPU 资源等。Impala 使用多种机制来控制 Impala 查询，MapReduce 作业和其他类型的作业共同协调使用集群资源：

- Impala 准入控制功能是一个轻量级的，分布式资源控制机制。它可以限制查询使用的内存段大小，并行执行的查询的数量等。如果查询请求到来时，系统已经达到了并行查询的最大数量，新的查询请求会被放入等待队列。当其他查询执行完成，释放了相关资源后，等待队列中的查询才能够被执行。另外，我们可以通过为不同的用户或者用户组分配不同的资源池来对可用的内存大小、并行查询的数量等进行不同的限制。
- 我们可以通过对 impalad 进程指定-mem_limit 选项来限制预留给查询的内存大小。这个参数仅限制被查询直接消耗的内存。而对于像元数据的缓存等其他内存，Impala 会单独进行分配。
- 在生产环境中，Cloudera 推荐为使用 Cloudera Manager 配置 Cgroup 机制来隔离资源的使用。

- 如果我们在 CDH5 上使用 Impala，则可以通过 Llama 来使用基于 YARN 的资源管理框架。

9.6 性能测试

测试 Impala 确保它使用了优化的配置。如果我们使用 Cloudera Manager 管理集群，则安装时它会自动地更新相关的配置信息；如果我们没有使用 Cloudera Manager 管理集群，则需要我们手动验证确保集群的配置正确。

1. 检查 Impala 配置信息

步骤 01 使用浏览器连接到 impalad 进程节点。

使用类似 http://hostname:port/varz 的 URL。其中 hostname 是要连接的 impalad 进程节点的主机名，端口号默认是 25000。

步骤 02 检查相关配置参数。

比如，如果我们要检查数据块本地性跟踪信息，我们需要确认参数 dfs.datanode.hdfs-blocks-metadata.enabled 为 TRUE。

2. 检查数据本地性

步骤 01 对一个多节点分布的数据集执行查询操作。比如，表 MyTable 是一个跨多个数据节点的分布表：

```
[impalad-host:21000] > SELECT COUNT (*) FROM MyTable
```

步骤 02 在查询执行完成之后，检查 Impala 日志信息。我们应该可以发现最近的日志中有类似信息：

```
Total remote scan volume = 0
```

这里的远程扫描可能标识了 impalad 进程没有正确的运行。引起这个问题的原因可能是在 DataNode 节点上没有运行 Impalad 进程，也可能是查询无法与 DataNode 上的 impalad 进程正确通信。

3. 问题排查步骤

步骤 01 连接到调试 web 服务器。该 web 服务器默认端口为 25000。在连接到的页面上我们可以看到集群中所有的 impalad 进程节点。确认是否所有 DataNode 上的 impalad 进程都正确启动。

步骤 02 确保 impalad 进程节点主机名能够被正确解析。在 impalad 进程启动时需要使用到主机

名。我们可以显式的通过--hostname 来设置主机名。

步骤 03 检查 statestored 进程是否正常运行。检查 statestored 进程的日志信息，确认所有的 impalad 进程节点都能正确地与 statestored 进程通信。

4. 检查日志信息

我们可以通过检查 impala 日志信息来确认是否发生了短路读，是否启用了块位置跟踪等。在检查日志之前，我们使用当前配置对一个很小的 HDFS 数据集执行一个简单查询。日志信息和它们的解释如下表所示：

Log Message	Interpretation
Unknown disk id. This will negatively affect performance. Check your hdfs settings to enable block location metadata	Tracking block locality is not enabled
Unable to load native-hadoop library for your platform... using builtin-java classes where applicable	Native checksumming is not enabled

5. 理解查询性能

为了从宏观上理解 Impala 查询的性能，我们可以不用真正的执行查询，直接使用 EXPLAIN 语句查看其执行计划。

为了从细节上了解 Impala 查询的吸能，我们可以在 impala-shell 中执行完查询之后使用 PROFILE 语句查看更详细的性能信息。通过 PROFILE 输出，我们可以了解到像内存消耗、CPU 消耗、IO 消耗、网络消耗等更详细的信息。

9.7 使用 EXPLAIN 信息

EXPLAIN 语句为我们提供了一个查询执行的逻辑步骤，包括怎样将查询分布到多个节点上，各节点之前怎样交换中间结果以及产生最终结果等。我们可以通过这些信息初步判断查询执行的方式是否高效。

```
[impalad-host:21000] > explain select count(*) from customer_address;
+----------------------------------------------------------+
| Explain String |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=42.00MB VCores=1 |
| |
| 03:AGGREGATE [MERGE FINALIZE] |
| | output: sum(count(*)) |
```

```
| | |
| 02:EXCHANGE [PARTITION=UNPARTITIONED] |
| | |
| 01:AGGREGATE |
| | output: count(*) |
| | |
| 00:SCAN HDFS [default.customer_address] |
| partitions=1/1 size=5.25MB |
+----------------------------------------------------------+
```

我们应当自底向上读输出结果：

（1）最后一部分内容展示了像读取的总数据量等底层的详细信息。通过读取的数据量，我们可以判断分区策略是否有效，并结合集群大小预估读取这些数据需要的时间等。

（2）我们可以看到操作是否被 Impala 不同的节点并行执行。

（3）我们可以从更高级别看到中间结果在不同节点间的流向。

（4）通过配置 EXPLAIN_LEVEL 参数，我们可以了解到更详细的输出信息。我们可以把这个参数由 0 改为 1 来确认统计信息是否存在，估算查询要消耗的资源等。

9.8 使用 PROFILE 信息

在 impala-shell 中，我们使用 PROFILE 语句可以输出最近执行的查询的更详细更底层的信息。与执行计划不同的是，查询必须执行完成后，才能使用 PROFILE 输出信息。它包括了执行该查询每个节点读取的物理字节数，使用的最大内存量等信息。通过这些信息，我们可以判断查询是 IO 消耗型的，CPU 消耗型的，网络消耗型的，或者受性能低下节点的影响，从而可以检查某些推荐的配置是否生效等。

通过 EXPLAIN_LEVEL 也可以控制通过 PROFILE 输出的执行计划的详细程度。

如下示例展示了在一个伪分布式集群中 PROFILE 的输出信息：

```
[hadoop-cs1:21000] > profile;
Query Runtime Profile:
Query (id=6540a03d4bee0691:4963d6269b210ebd):
Summary:
Session ID: ea4a197f1c7bf858:c74e66f72e3a33ba
Session Type: BEESWAX
Start Time: 2013-12-02 17:10:30.263067000
End Time: 2013-12-02 17:10:50.932044000
```

```
Query Type: QUERY
Query State: FINISHED
Query Status: OK
Impala Version: impalad version 1.2.1 RELEASE (build
edb5af1bcad63d410bc5d47cc203df3a880e9324)
User: cloudera
Network Address: 127.0.0.1:49161
Default Db: stats_testing
Sql Statement: select t1.s, t2.s from t1 join t2 on (t1.id = t2.parent)
Plan:
----------------
Estimated Per-Host Requirements: Memory=2.09GB VCores=2
PLAN FRAGMENT 0
PARTITION: UNPARTITIONED
4:EXCHANGE
cardinality: unavailable
per-host memory: unavailable
tuple ids: 0 1
PLAN FRAGMENT 1
PARTITION: RANDOM
STREAM DATA SINK
EXCHANGE ID: 4
UNPARTITIONED
2:HASH JOIN
| join op: INNER JOIN (BROADCAST)
| hash predicates:
| t1.id = t2.parent
| cardinality: unavailable
| per-host memory: 2.00GB
| tuple ids: 0 1
|
|----3:EXCHANGE
| cardinality: unavailable
| per-host memory: 0B
| tuple ids: 1
|
0:SCAN HDFS
table=stats_testing.t1 #partitions=1/1 size=33B
table stats: unavailable
```

```
column stats: unavailable
cardinality: unavailable
per-host memory: 32.00MB
tuple ids: 0
PLAN FRAGMENT 2
PARTITION: RANDOM
STREAM DATA SINK
EXCHANGE ID: 3
UNPARTITIONED
1:SCAN HDFS
table=stats_testing.t2 #partitions=1/1 size=960.00KB
table stats: unavailable
column stats: unavailable
cardinality: unavailable
per-host memory: 96.00MB
tuple ids: 1
----------------
Query Timeline: 20s670ms
- Start execution: 2.559ms (2.559ms)
- Planning finished: 23.587ms (21.27ms)
- Rows available: 666.199ms (642.612ms)
- First row fetched: 668.919ms (2.719ms)
- Unregister query: 20s668ms (20s000ms)
ImpalaServer:
- ClientFetchWaitTimer: 19s637ms
- RowMaterializationTimer: 167.121ms
Execution Profile 6540a03d4bee0691:4963d6269b210ebd:(Active: 837.815ms, %
non-child:
0.00%)
Per Node Peak Memory Usage: impala-1.example.com:22000(7.42 MB)
- FinalizationTimer: 0ns
Coordinator Fragment:(Active: 195.198ms, % non-child: 0.00%)
MemoryUsage(500.0ms): 16.00 KB, 7.42 MB, 7.33 MB, 7.10 MB, 6.94 MB, 6.71 MB, 6.56
MB, 6.40 MB, 6.17 MB, 6.02 MB, 5.79 MB, 5.63 MB, 5.48 MB, 5.25 MB, 5.09 MB, 4.86
MB,
4.71 MB, 4.47 MB, 4.32 MB, 4.09 MB, 3.93 MB, 3.78 MB, 3.55 MB, 3.39 MB, 3.16 MB,
3.01
MB, 2.78 MB, 2.62 MB, 2.39 MB, 2.24 MB, 2.08 MB, 1.85 MB, 1.70 MB, 1.54 MB, 1.31
MB,
```

```
1.16 MB, 948.00 KB, 790.00 KB, 553.00 KB, 395.00 KB, 237.00 KB
ThreadUsage(500.0ms): 1
- AverageThreadTokens: 1.00
- PeakMemoryUsage: 7.42 MB
- PrepareTime: 36.144us
- RowsProduced: 98.30K (98304)
- TotalCpuTime: 20s449ms
- TotalNetworkWaitTime: 191.630ms
- TotalStorageWaitTime: 0ns
CodeGen:(Active: 150.679ms, % non-child: 77.19%)
- CodegenTime: 0ns
- CompileTime: 139.503ms
- LoadTime: 10.7ms
- ModuleFileSize: 95.27 KB
EXCHANGE_NODE (id=4):(Active: 194.858ms, % non-child: 99.83%)
- BytesReceived: 2.33 MB
- ConvertRowBatchTime: 2.732ms
- DataArrivalWaitTime: 191.118ms
- DeserializeRowBatchTimer: 14.943ms
- FirstBatchArrivalWaitTime: 191.117ms
- PeakMemoryUsage: 7.41 MB
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 504.49 K/sec
- SendersBlockedTimer: 0ns
- SendersBlockedTotalTimer(*): 0ns
Averaged Fragment 1:(Active: 442.360ms, % non-child: 0.00%)
split sizes: min: 33.00 B, max: 33.00 B, avg: 33.00 B, stddev: 0.00
completion times: min:443.720ms max:443.720ms mean: 443.720ms stddev:0ns
execution rates: min:74.00 B/sec max:74.00 B/sec mean:74.00 B/sec stddev:0.00
/sec
num instances: 1
- AverageThreadTokens: 1.00
- PeakMemoryUsage: 6.06 MB
- PrepareTime: 7.291ms
- RowsProduced: 98.30K (98304)
- TotalCpuTime: 784.259ms
- TotalNetworkWaitTime: 388.818ms
- TotalStorageWaitTime: 3.934ms
CodeGen:(Active: 312.862ms, % non-child: 70.73%)
```

```
- CodegenTime: 2.669ms
- CompileTime: 302.467ms
- LoadTime: 9.231ms
- ModuleFileSize: 95.27 KB
DataStreamSender (dst_id=4):(Active: 80.63ms, % non-child: 18.10%)
- BytesSent: 2.33 MB
- NetworkThroughput(*): 35.89 MB/sec
- OverallThroughput: 29.06 MB/sec
- PeakMemoryUsage: 5.33 KB
- SerializeBatchTime: 26.487ms
- ThriftTransmitTime(*): 64.814ms
- UncompressedRowBatchSize: 6.66 MB
HASH_JOIN_NODE (id=2):(Active: 362.25ms, % non-child: 3.92%)
- BuildBuckets: 1.02K (1024)
- BuildRows: 98.30K (98304)
- BuildTime: 12.622ms
- LoadFactor: 0.00
- PeakMemoryUsage: 6.02 MB
- ProbeRows: 3
- ProbeTime: 3.579ms
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 271.54 K/sec
EXCHANGE_NODE (id=3):(Active: 344.680ms, % non-child: 77.92%)
- BytesReceived: 1.15 MB
- ConvertRowBatchTime: 2.792ms
- DataArrivalWaitTime: 339.936ms
- DeserializeRowBatchTimer: 9.910ms
- FirstBatchArrivalWaitTime: 199.474ms
- PeakMemoryUsage: 156.00 KB
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 285.20 K/sec
- SendersBlockedTimer: 0ns
- SendersBlockedTotalTimer(*): 0ns
HDFS_SCAN_NODE (id=0):(Active: 13.616us, % non-child: 0.00%)
- AverageHdfsReadThreadConcurrency: 0.00
- AverageScannerThreadConcurrency: 0.00
- BytesRead: 33.00 B
- BytesReadLocal: 33.00 B
- BytesReadShortCircuit: 33.00 B
```

```
- NumDisksAccessed: 1
- NumScannerThreadsStarted: 1
- PeakMemoryUsage: 46.00 KB
- PerReadThreadRawHdfsThroughput: 287.52 KB/sec
- RowsRead: 3
- RowsReturned: 3
- RowsReturnedRate: 220.33 K/sec
- ScanRangesComplete: 1
- ScannerThreadsInvoluntaryContextSwitches: 26
- ScannerThreadsTotalWallClockTime: 55.199ms
- DelimiterParseTime: 2.463us
- MaterializeTupleTime(*): 1.226us
- ScannerThreadsSysTime: 0ns
- ScannerThreadsUserTime: 42.993ms
- ScannerThreadsVoluntaryContextSwitches: 1
- TotalRawHdfsReadTime(*): 112.86us
- TotalReadThroughput: 0.00 /sec
Averaged Fragment 2:(Active: 190.120ms, % non-child: 0.00%)
split sizes: min: 960.00 KB, max: 960.00 KB, avg: 960.00 KB, stddev: 0.00
completion times: min:191.736ms max:191.736ms mean: 191.736ms stddev:0ns
execution rates: min:4.89 MB/sec max:4.89 MB/sec mean:4.89 MB/sec stddev:0.00
/sec
num instances: 1
- AverageThreadTokens: 0.00
- PeakMemoryUsage: 906.33 KB
- PrepareTime: 3.67ms
- RowsProduced: 98.30K (98304)
- TotalCpuTime: 403.351ms
- TotalNetworkWaitTime: 34.999ms
- TotalStorageWaitTime: 108.675ms
CodeGen:(Active: 162.57ms, % non-child: 85.24%)
- CodegenTime: 3.133ms
- CompileTime: 148.316ms
- LoadTime: 12.317ms
- ModuleFileSize: 95.27 KB
DataStreamSender (dst_id=3):(Active: 70.620ms, % non-child: 37.14%)
- BytesSent: 1.15 MB
- NetworkThroughput(*): 23.30 MB/sec
- OverallThroughput: 16.23 MB/sec
```

```
- PeakMemoryUsage: 5.33 KB
- SerializeBatchTime: 22.69ms
- ThriftTransmitTime(*): 49.178ms
- UncompressedRowBatchSize: 3.28 MB
HDFS_SCAN_NODE (id=1):(Active: 118.839ms, % non-child: 62.51%)
- AverageHdfsReadThreadConcurrency: 0.00
- AverageScannerThreadConcurrency: 0.00
- BytesRead: 960.00 KB
- BytesReadLocal: 960.00 KB
- BytesReadShortCircuit: 960.00 KB
- NumDisksAccessed: 1
- NumScannerThreadsStarted: 1
- PeakMemoryUsage: 869.00 KB
- PerReadThreadRawHdfsThroughput: 130.21 MB/sec
- RowsRead: 98.30K (98304)
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 827.20 K/sec
- ScanRangesComplete: 15
- ScannerThreadsInvoluntaryContextSwitches: 34
- ScannerThreadsTotalWallClockTime: 189.774ms
- DelimiterParseTime: 15.703ms
- MaterializeTupleTime(*): 3.419ms
- ScannerThreadsSysTime: 1.999ms
- ScannerThreadsUserTime: 44.993ms
- ScannerThreadsVoluntaryContextSwitches: 118
- TotalRawHdfsReadTime(*): 7.199ms
- TotalReadThroughput: 0.00 /sec
Fragment 1:
Instance 6540a03d4bee0691:4963d6269b210ebf
(host=impala-1.example.com:22000):(Active: 442.360ms, % non-child: 0.00%)
Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:1/33.00 B
MemoryUsage(500.0ms): 69.33 KB
ThreadUsage(500.0ms): 1
- AverageThreadTokens: 1.00
- PeakMemoryUsage: 6.06 MB
- PrepareTime: 7.291ms
- RowsProduced: 98.30K (98304)
- TotalCpuTime: 784.259ms
- TotalNetworkWaitTime: 388.818ms
```

```
- TotalStorageWaitTime: 3.934ms
CodeGen:(Active: 312.862ms, % non-child: 70.73%)
- CodegenTime: 2.669ms
- CompileTime: 302.467ms
- LoadTime: 9.231ms
- ModuleFileSize: 95.27 KB
DataStreamSender (dst_id=4):(Active: 80.63ms, % non-child: 18.10%)
- BytesSent: 2.33 MB
- NetworkThroughput(*): 35.89 MB/sec
- OverallThroughput: 29.06 MB/sec
- PeakMemoryUsage: 5.33 KB
- SerializeBatchTime: 26.487ms
- ThriftTransmitTime(*): 64.814ms
- UncompressedRowBatchSize: 6.66 MB
HASH_JOIN_NODE (id=2):(Active: 362.25ms, % non-child: 3.92%)
ExecOption: Build Side Codegen Enabled, Probe Side Codegen Enabled, Hash
Table Built Asynchronously
- BuildBuckets: 1.02K (1024)
- BuildRows: 98.30K (98304)
- BuildTime: 12.622ms
- LoadFactor: 0.00
- PeakMemoryUsage: 6.02 MB
- ProbeRows: 3
- ProbeTime: 3.579ms
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 271.54 K/sec
EXCHANGE_NODE (id=3):(Active: 344.680ms, % non-child: 77.92%)
- BytesReceived: 1.15 MB
- ConvertRowBatchTime: 2.792ms
- DataArrivalWaitTime: 339.936ms
- DeserializeRowBatchTimer: 9.910ms
- FirstBatchArrivalWaitTime: 199.474ms
- PeakMemoryUsage: 156.00 KB
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 285.20 K/sec
- SendersBlockedTimer: 0ns
- SendersBlockedTotalTimer(*): 0ns
HDFS_SCAN_NODE (id=0):(Active: 13.616us, % non-child: 0.00%)
Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:1/33.00 B
```

```
Hdfs Read Thread Concurrency Bucket: 0:0% 1:0%
File Formats: TEXT/NONE:1
ExecOption: Codegen enabled: 1 out of 1
- AverageHdfsReadThreadConcurrency: 0.00
- AverageScannerThreadConcurrency: 0.00
- BytesRead: 33.00 B
- BytesReadLocal: 33.00 B
- BytesReadShortCircuit: 33.00 B
- NumDisksAccessed: 1
- NumScannerThreadsStarted: 1
- PeakMemoryUsage: 46.00 KB
- PerReadThreadRawHdfsThroughput: 287.52 KB/sec
- RowsRead: 3
- RowsReturned: 3
- RowsReturnedRate: 220.33 K/sec
- ScanRangesComplete: 1
- ScannerThreadsInvoluntaryContextSwitches: 26
- ScannerThreadsTotalWallClockTime: 55.199ms
- DelimiterParseTime: 2.463us
- MaterializeTupleTime(*): 1.226us
- ScannerThreadsSysTime: 0ns
- ScannerThreadsUserTime: 42.993ms
- ScannerThreadsVoluntaryContextSwitches: 1
- TotalRawHdfsReadTime(*): 112.86us
- TotalReadThroughput: 0.00 /sec
Fragment 2:
Instance 6540a03d4bee0691:4963d6269b210ec0
(host=impala-1.example.com:22000):(Active: 190.120ms, % non-child: 0.00%)
Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:15/960.00 KB
- AverageThreadTokens: 0.00
- PeakMemoryUsage: 906.33 KB
- PrepareTime: 3.67ms
- RowsProduced: 98.30K (98304)
- TotalCpuTime: 403.351ms
- TotalNetworkWaitTime: 34.999ms
- TotalStorageWaitTime: 108.675ms
CodeGen:(Active: 162.57ms, % non-child: 85.24%)
- CodegenTime: 3.133ms
- CompileTime: 148.316ms
```

```
- LoadTime: 12.317ms
- ModuleFileSize: 95.27 KB
DataStreamSender (dst_id=3):(Active: 70.620ms, % non-child: 37.14%)
- BytesSent: 1.15 MB
- NetworkThroughput(*): 23.30 MB/sec
- OverallThroughput: 16.23 MB/sec
- PeakMemoryUsage: 5.33 KB
- SerializeBatchTime: 22.69ms
- ThriftTransmitTime(*): 49.178ms
- UncompressedRowBatchSize: 3.28 MB
HDFS_SCAN_NODE (id=1):(Active: 118.839ms, % non-child: 62.51%)
Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:15/960.00 KB
Hdfs Read Thread Concurrency Bucket: 0:0% 1:0%
File Formats: TEXT/NONE:15
ExecOption: Codegen enabled: 15 out of 15
- AverageHdfsReadThreadConcurrency: 0.00
- AverageScannerThreadConcurrency: 0.00
- BytesRead: 960.00 KB
- BytesReadLocal: 960.00 KB
- BytesReadShortCircuit: 960.00 KB
- NumDisksAccessed: 1
- NumScannerThreadsStarted: 1
- PeakMemoryUsage: 869.00 KB
- PerReadThreadRawHdfsThroughput: 130.21 MB/sec
- RowsRead: 98.30K (98304)
- RowsReturned: 98.30K (98304)
- RowsReturnedRate: 827.20 K/sec
- ScanRangesComplete: 15
- ScannerThreadsInvoluntaryContextSwitches: 34
- ScannerThreadsTotalWallClockTime: 189.774ms
- DelimiterParseTime: 15.703ms
- MaterializeTupleTime(*): 3.419ms
- ScannerThreadsSysTime: 1.999ms
- ScannerThreadsUserTime: 44.993ms
- ScannerThreadsVoluntaryContextSwitches: 118
- TotalRawHdfsReadTime(*): 7.199ms
- TotalReadThroughput: 0.00 /sec
```

第 10 章

◂ Impala设计原则与应用案例 ▸

根据之前章节的学习，我们发现 Impala 是一个基于 SQL，以 HDFS 为存储，大规模 MPP 运算引擎。那么如何将 Impala 应用到企业环境中呢？我们将在本章第 1 节中说明了使用 Impala 需要遵循的设计原则，在第 2 节中通过实际案例来解释这些设计原则。

10.1 设计原则

1. 硬件规划

如果对一个已有的 CDH 集群使用 Impala 技术，在硬件层面上我们最需要关注的点就是：我们需要 Impala 处理的最大的表或者分区容量。因为 Impala 要将数据全部读入内存才进行运算，我们必须要保证内存能够装载下所需要的表或者分区的数据。一般情况下，由于原有的集群是以基于磁盘的操作为主，需要为各数据节点添加足够的内存。

2. 模型设计

在传统的应用开发流程中，我们需要对数据库进行物理设计，这里的模型设计与传统的物理设计类似。当然，具体需要设计成什么样，完全取决于具体的应用场景。

- 存储方式：未来的 Impala 表使用什么类型的数据文件存储？数据量如果较小，可以使用数据文件的原有格式。如果数据量非常庞大，而且对数据的查询是基于少量列进行的，则强烈建议使用 Parquet 方式存储。
- 分区方式：针对 Impala 表进行的频度最高的查询依据什么条件？条件列的区分度如何？如果按条件列来进行分区，单个分区的数据量有多大？是否需要使用几个列的组合来进行分区？选择合适的分区，能够大大提升查询的效率。
- 内部表&外部表：使用外部表，不需要数据复制的过程，但不能改变存储的数据文件的格式。使用内部表，在各方面的控制上更为灵活，但是需要将外部的数据复制到 Impala 内部的处理过程。
- 资源控制：Hadoop 的其他类型的作业是否会与 Impala 的查询并行执行？使用 YARN 来管理 Impala 的资源控制，还是 Impala 使用自身的机制进行管理？

3. 数据加载

如果是现有的 HDFS 上已经有了相应的数据文件，本步骤可以忽略。

如果现有的 Hadoop 集群是新建的，仅供 Impala 来使用，那可能就需要从外部数据源加载数据了。

- 文件加载：如果外部数据源有现成的 Impala 支持的数据文件，可以通过 HDFS 文件系统的命令，或者基于 WebHDFS 和 HttpFS 的 API 来编写代码实现。
- 结构化数据加载：如果我们需要把传统关系型数据库中的数据加载到 HDFS 上，可以通过 Sqoop 来完成。我们可以使用 Sqoop 将关系型数据库中全量，或者增量的数据（依据一定的条件）加载到 HDFS 上。Sqoop 配置好之后，它会以 MapReduce 的方式将数据加载到 HDFS 上。事实上，Sqoop 并不支持所有 Impala 支持的文件格式，如果我们需要的数据文件格式 Sqoop 不支持，我们还需要进行二次转换。
- 基于事件的数据加载：对于源端的数据变化必须很快反映到目标数据中的情况，我们可以考虑使用 Flume 来实现。Flume 具备从控制台、RPC、text、tail、syslog、exec 等数据源搜集数据的能力。而且 Flume 还提供了数据收集和传输的高可用性。

4. 数据处理

利用 Impala 提供的强大的 SQL 功能，根据应用不同的需求对数据进行查询处理。

5. 数据返回

处理完成的数据，我们可以通过 JDBC 接口来调用，比如，可以给 BI 工具做报表展现使用。另外，我们也可以将处理结果返回到传统的关系型数据库供用户使用。

10.2 应用案例

1. 硬件规划

某保险公司原有 CDH4.6 集群有数据节点 35 个，数据节点 2*8Core CPU，内存 32GB，硬盘 2*2TB。经调查，最大单表的数据量为 3TB 左右。那么平均每个节点的内存量应该为：

```
3TB/35=86GB
```

考虑到可扩展及其他因素，增加每个节点的内存到 96GB，相当于每个节点再加 10GB。

2. 模型设计

模型设计使用下面方式：

- 存储方式：使用 Parquet 文件存储。

- 分区方式：表依据最常用的 ACTYR、GEOST、ALINE、COMPNY 字段分区。
- 内部表&外部表：原有数据使用 CSV 方式存储，需要使用 Impala 内部表。
- 资源控制：将 Impala 即席查询任务与 Hadoop 其他批处理理任务错峰执行，使用 Impala 自身的资源控制机制。

```
DROP TABLE IF EXISTS default.datasetp_2010_2013;
CREATE TABLE IF NOT EXISTS default.datasetp_2010_2013
(
PROC_MONTH STRING, PROC_YEAR STRING, POLCT STRING, STATST STRING, [800+ column definitions removed for brevity]
EEXP REAL, EPREM REAL, EXP REAL, PREM REAL
)
PARTITIONED BY (ACTYR STRING, GEOST STRING, ALINE STRING, COMPNY STRING)
STORED AS PARQUET;
```

3. 数据加载

原有 CSV 文件已经存储在 HDFS 上，使用 LOAD DATA 先将数据加载到 Impala 外部表，再通过外部表将数据复制到基于 Parquet 的 Impala 内部表，完成数据转换。

```
DROP TABLE IF EXISTS default.dataset_2010_2013;
CREATE EXTERNAL TABLE IF NOT EXISTS default.dataset_2010_2013
(
PROC_MONTH STRING, PROC_YEAR STRING, POLCT STRING, COMPNY STRING, GEOST STRING,STATST STRING, [800+ column definitions removed for brevity]
EEXP REAL, EPREM REAL, EXP REAL, PREM REAL
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/user/drake/cdf_impala';

INSERT INTO default.datasetp_2010_2013
(
PROC_MONTH, PROC_YEAR, POLCT, CLINE, STATST, ALLORO, ACTMO, ACTDA, [800+ column definitions removed for brevity]
)
PARTITION (ACTYR, GEOST, ALINE, COMPNY)
SELECT
PROC_MONTH, PROC_YEAR, POLCT, CLINE, STATST, ALLORO, ACTMO, ACTDA, [800+ column definitions removed for brevity]
```

```
ACTYR, GEOST, ALINE, COMPNY
FROM default.dataset_2010_2013;
```

4. 数据处理

完成按照不同维度对数据进行实时统计分析。

```
[datanode01:21000] > explain select count(*) from default.datasetp_2010_2013;
Query: explain select count(*) from default.datasetp_2010_2013
+----------------------------------------------------------------------+
| Explain String                                                       |
+----------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                      |
|   PARTITION: UNPARTITIONED                                           |
|                                                                      |
|   3:AGGREGATE (merge finalize)                                       |
|   |  output: SUM(COUNT(*))                                           |
|   |                                                                  |
|   2:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 1                                                      |
|   PARTITION: RANDOM                                                  |
|                                                                      |
|   STREAM DATA SINK                                                   |
|     EXCHANGE ID: 2                                                   |
|     UNPARTITIONED                                                    |
|                                                                      |
|   1:AGGREGATE                                                        |
|   |  output: COUNT(*)                                                |
|   |                                                                  |
|   0:SCAN HDFS                                                        |
|      table=default.datasetp_2010_2013 #partitions=787/787 size=105.90GB |
+----------------------------------------------------------------------+
Returned 20 row(s) in 0.07s
[datanode01:21000] > select count(*) from default.datasetp_2010_2013;
Query: select count(*) from default.datasetp_2010_2013
+------------+
| count(*)   |
+------------+
| 1168654867 |
```

```
+------------+
Returned 1 row(s) in 1.94s
```

从上面最后一行可以看出来，经过了良好的模式设计，对整个表的 count，时间只需要 1.94s。再看下面的查询示例：

```
[datanode01:21000] > explain select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 GROUP BY ACTYR;
Query: explain select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 GROUP BY ACTYR
+----------------------------------------------------------------------+
| Explain String                                                       |
+----------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                      |
|   PARTITION: UNPARTITIONED                                           |
|                                                                      |
|   5:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 1                                                      |
|   PARTITION: HASH_PARTITIONED: ACTYR                                 |
|                                                                      |
|   STREAM DATA SINK                                                   |
|     EXCHANGE ID: 5                                                   |
|     UNPARTITIONED                                                    |
|                                                                      |
|   2:AGGREGATE (merge finalize)                                       |
|   |  output: COUNT(GEOST)                                            |
|   |  group by: ACTYR                                                 |
|   |                                                                  |
|   4:AGGREGATE (merge)                                                |
|   |  group by: ACTYR, GEOST                                          |
|   |                                                                  |
|   3:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 2                                                      |
|   PARTITION: RANDOM                                                  |
|                                                                      |
|   STREAM DATA SINK                                                   |
|     EXCHANGE ID: 3                                                   |
|     HASH_PARTITIONED: ACTYR                                          |
```

```
|                                                                        |
|   1:AGGREGATE                                                          |
|   |  group by: ACTYR, GEOST                                            |
|   |                                                                    |
|   0:SCAN HDFS                                                          |
|      table=default.datasetp_2010_2013 #partitions=787/787 size=105.90GB |
+------------------------------------------------------------------------+
Returned 33 row(s) in 0.06s
[datanode01:21000] > select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 GROUP BY ACTYR;
Query: select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 GROUP BY ACTYR
+-----------------------+-------+
| count(distinct geost) | actyr |
+-----------------------+-------+
| 42                    | 112   |
| 42                    | 110   |
| 42                    | 113   |
| 42                    | 111   |
+-----------------------+-------+
Returned 4 row(s) in 8.69s
```

从上面最后一行可以看出来，按照地理位置进行年份的分组统计，查询时间需要 8.69s。再看下面的查询示例：

```
[datanode01:21000] > explain select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 WHERE ACTYR = '111' GROUP BY ACTYR;
Query: explain select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 WHERE ACTYR = '111' GROUP BY ACTYR
+----------------------------------------------------------------------+
| Explain String                                                       |
+----------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                      |
|   PARTITION: UNPARTITIONED                                           |
|                                                                      |
|   5:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 1                                                      |
|   PARTITION: HASH_PARTITIONED: ACTYR                                 |
|                                                                      |
```

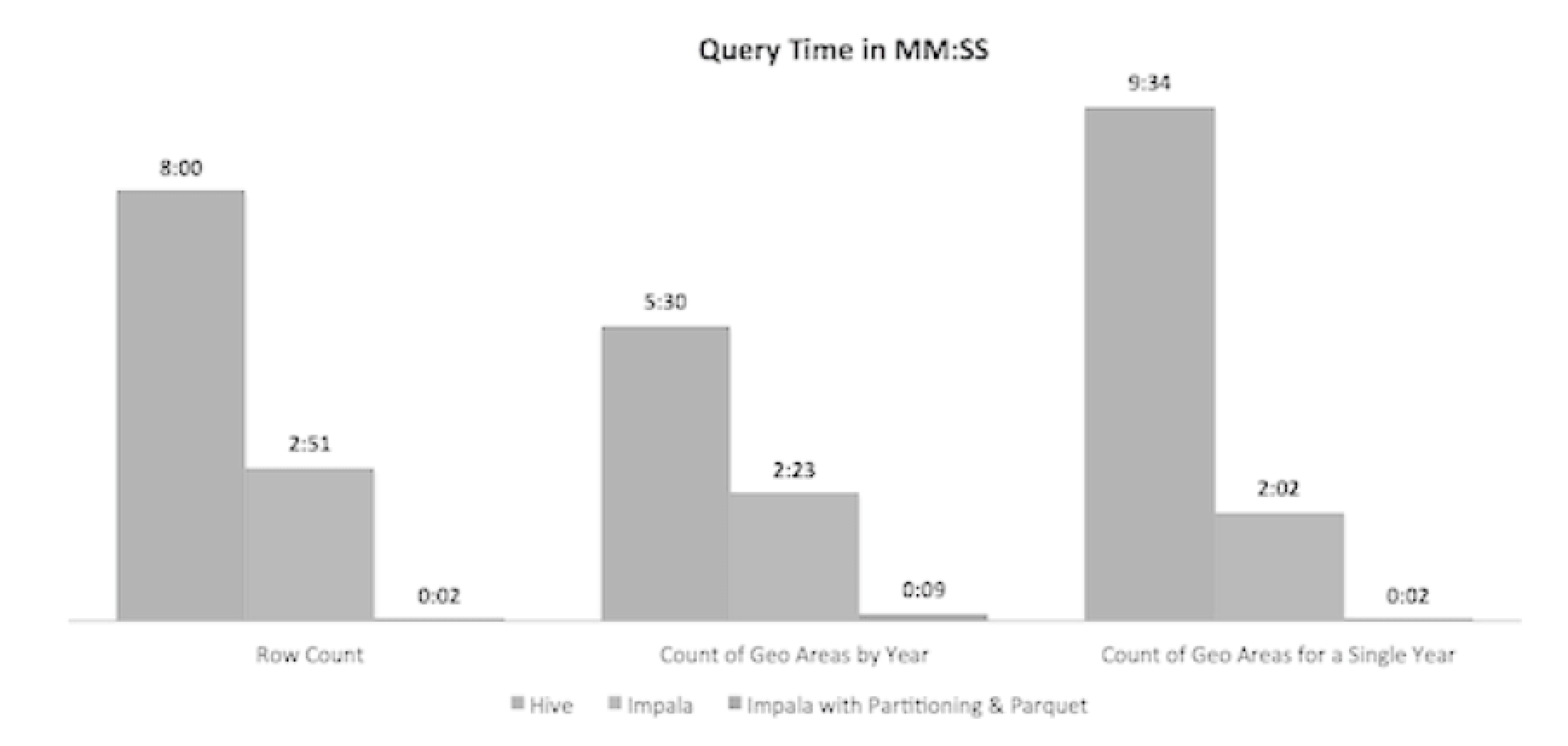
```
|   STREAM DATA SINK                                                  |
|     EXCHANGE ID: 5                                                  |
|     UNPARTITIONED                                                   |
|                                                                     |
|   2:AGGREGATE (merge finalize)                                      |
|   |  output: COUNT(GEOST)                                           |
|   |  group by: ACTYR                                                |
|   |                                                                 |
|   4:AGGREGATE (merge)                                               |
|   |  group by: ACTYR, GEOST                                         |
|   |                                                                 |
|   3:EXCHANGE                                                        |
|                                                                     |
| PLAN FRAGMENT 2                                                     |
|   PARTITION: RANDOM                                                 |
|                                                                     |
|   STREAM DATA SINK                                                  |
|     EXCHANGE ID: 3                                                  |
|     HASH_PARTITIONED: ACTYR                                         |
|                                                                     |
|   1:AGGREGATE                                                       |
|   |  group by: ACTYR, GEOST                                         |
|   |                                                                 |
|   0:SCAN HDFS                                                       |
|      table=default.datasetp_2010_2013 #partitions=198/787 size=24.25GB |
+---------------------------------------------------------------------+
Returned 33 row(s) in 0.22s
[datanode01:21000] > select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 WHERE ACTYR = '111' GROUP BY ACTYR;
Query: select count(distinct GEOST), ACTYR from default.datasetp_2010_2013 WHERE ACTYR = '111' GROUP BY ACTYR
+-----------------------+-------+
| count(distinct geost) | actyr |
+-----------------------+-------+
| 42                    | 111   |
+-----------------------+-------+
Returned 1 row(s) in 1.77s
```

从上面最后一行可以看出来，针对特定年份的对地理位置的分组统计仅需要 1.77s。

5. 数据返回

将查询到的结果通过 Tableau 进行报表展示。

如数据处理中的三个场景，如果直接用 Hive 处理，或者不经过模式设计，只简单地将数据加载到 Impala 中查询，与严格按照规划步骤设计后处理的查询结果都相去甚远，对比图如下：



通过上图可以看到，单单从 Hive 到 Impala，不做任何其他改变，查询速度平均提升了三分之二以上，如果经过良好的模式设计，执行速度有数量级的提升。从这一个侧面也足以证明 Impala 查询能力的优秀。